Introduction To Rogue

Ryan Herbst

(rherbst@slac.stanford.edu)

SLAC TID-AIR
Technology Innovation Directorate
Advanced Instrumentation for Research Division

U.S. DEPARTMENT OF ENERGY
Office of Science

SLAC NATIONAL ACCELERATOR LABORATORY

Goals Of Rogue

SLAC

- Provide a system to facilitate hardware development and intermediate daq systems for interfacing to hardware
- Support a number of hardware & software interface technologies
  - Including ones that don't yet exist
- Easy to understand mechanisms for connecting independent management and data processing modules together using a set of well defined, easy to understand interfaces
- Allow data paths to exist in independent high performance threads
  - While also allowing data access to python for visualization
- Flexible structure for creating a hierarchy of system components
  - Independent of the network and hardware hierarchies
- Internode coordination and management of multi-server systems (RCE clusters)
- Support data file writing & configuration archiving
- Interface independently to a number of management layers, sometimes in parallel
  - EPICS,
  - CODA
  - Ignition (mysql)
  - EuDaq

Why Rogue?

SLAC

- Original proposed name was SCAMP
  - System Control & Acquisition with Modular Python
- Dictionary definition of SCAMP::

  a person, especially a child, who is mischievous in a likable or amusing way.

  a wicked or worthless person; a rogue.
- The term rogue stood out and was chosen as the system name for development
- Rumors that the first git repository was called rouge due to a email typo, that was mistakenly cut and pasted, can not be confirmed!

Helpful Links

SLAC

- Rogue is on github:
  - https://github.com/slaclab/rogue
- SLAC email list for rogue announcements:
  - https://listserv.slac.stanford.edu/cgi-bin/wa?A0=AIR-ROGUE-USERS
  - Please subscribe if you are a user of rogue
- Rogue-daq channel on SLACK
  - https://slac.slack.com/#rogue-daq
  - Developers are usually available
- SLAC JIRA for the rogue project
  - https://jira.slac.stanford.edu/projects/ESROGUE/summary
  - Please submit tickets if you are seeing any bugs
- Questions:
  - Either find us on slack or email:
    - rherbst@slac.stanford.edu
    - bareese@slac.stanford.edu

Structure Of Rogue

SLAC

- Mixed C++/Python codebase using boost::python library
  - C++ base classes expose most of their methods to Python layer
  - C++ threads are used for data path and underlying communication mechanisms
  - Allows rapid development in Python with ability to drop into C++ for performance
  - **Most development with Rogue is done in Python**

- Low level (C++) base structures
  - **Stream** interface for bulk data movement and asynchronous messages
    - Rogue::interfaces::stream
  - **Memory** interface for register access
    - rogue::interfaces::memory

- Higher level (Python) interface for organizing systems
  - Tree structure for hierarchical organization
  - **Devices** contain **Variables**, **Commands**, and other **Devices**
  - **Variables** describe registers - Address, data type, etc
  - **Commands** describe common sequences of operations on a **Device**

- Requirements:
  - Python 3.6.1 or newer
  - Boost 1.64.0 or newer
  - See Readme.md for more details

AFS Based Rogue Resources

SLAC

- An up to date version of rogue, along with tagged releases can be found in:

  /afs/slac.stanford.edu/g/reseng/rogue/

- Support libraries can be found in:

  /afs/slac.stanford.edu/g/reseng/boost/1.64.0
  /afs/slac.stanford.edu/g/reseng/boost/1.64.0

- To use the central rogue installation:

  source /afs/slac.stanford.edu/g/reseng/rogue/master/setup_env.csh
  or
  source /afs/slac.stanford.edu/g/reseng/rogue/master/setup_env.sh

- Additional libraries if you are using the zeromq (simulation link) or epics features of rogue:
  - These are already included in the afs rogue environment

    /afs/slac.stanford.edu/g/reseng/zeromq/4.2.0
    /afs/slac.stanford.edu/g/reseng/epics/base-R3-16-0

Rogue Namespaces

SLAC

- rogue : C++ classes exposed to python layer
  - interfaces: Interfaces for inter-module connections
    - stream: Streaming interfaces for bulk data & asynchronous messages
    - memory: Memory interfaces for hardware memory access
  - protocols: Protocol modules
    - srp: SLAC register protocol
    - rssi: Reliable SLAC streaming interface
    - packetizer: SLAC packetizer protocol
    - UDP: UDP interface
  - hardware: Hardware interface modules
    - exo: EXO hardware (PCI-Tem)
    - pgp: PGP Card support
    - rce: RCE streaming and memory mapped interfaces
    - data: Data processing PCI-Card development
  - utilities: Utilities for application support
    - fileio: File read/write support objects
    - prbs: Test data receive and generation
  - Other (at top level)
    - Logging: C++ logging and debug utility
    - SMemControl: Shared memory control interface
    - ApiWrapper: Wrapper for embedding pyrogue within a c++ daq system
    - Version: Version information
    - Support classes:
      - GIL acquire and release
      - Queues, etc

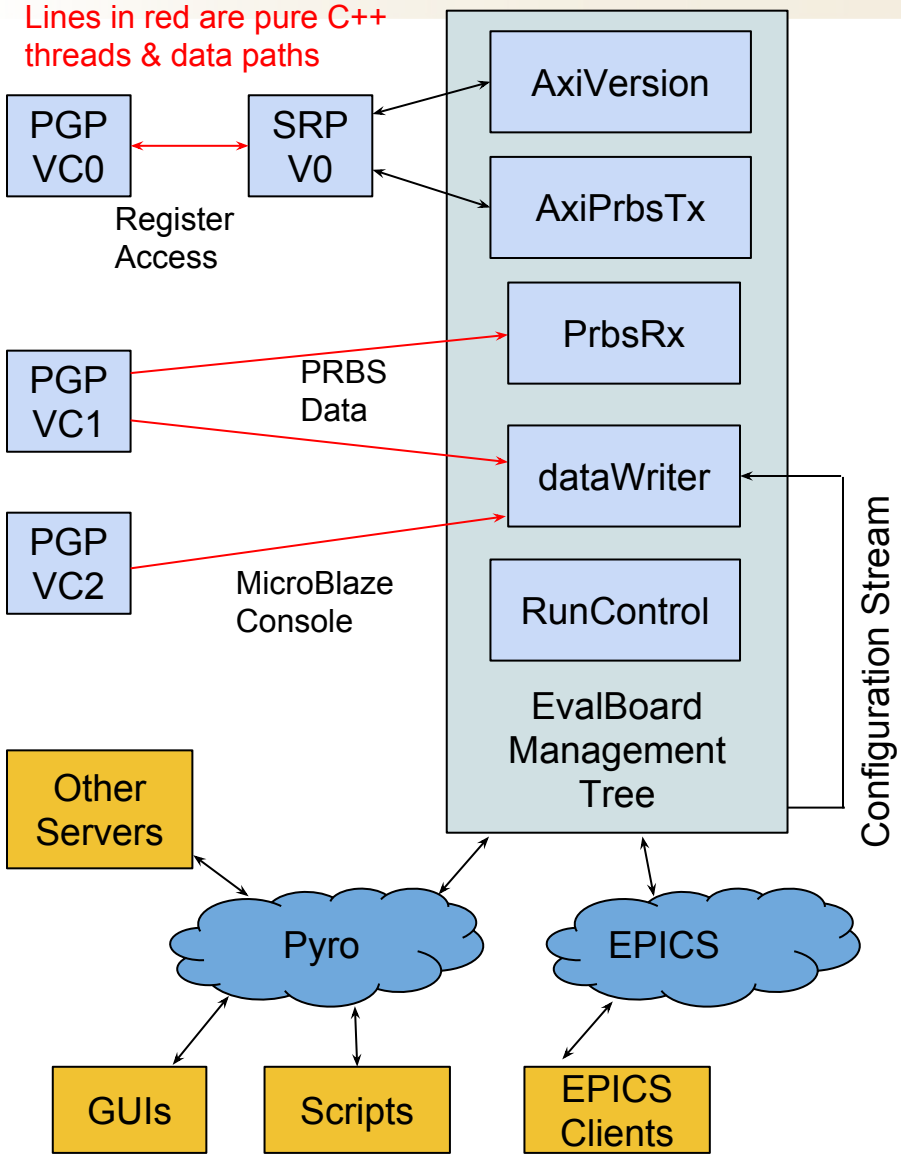
Rogue Namespaces

SLAC

- pyrogue: Pure python modules and wrappers
  - gui: GUI modules
  - utilities: Pure python utilities and management wrappers for rogue utilities
  - epics: Epics interface library
  - protocols: Wrappers for c++ protocol classes
  - simulation: Support for VHDL simulation interface
  - smem: Support for shared memory interface

- Most rogue classes appear immediately below pyrogue:
  - pyrogue.Node
  - pyrogue.Root
  - pyrogue.Device
  - pyrogue.LocalVariable
  - pyrogue.RemoteVariable
  - pyrogue.LinkVariable
  - pyrogue.LocalCommand
  - pyrogue.RemoteCommand

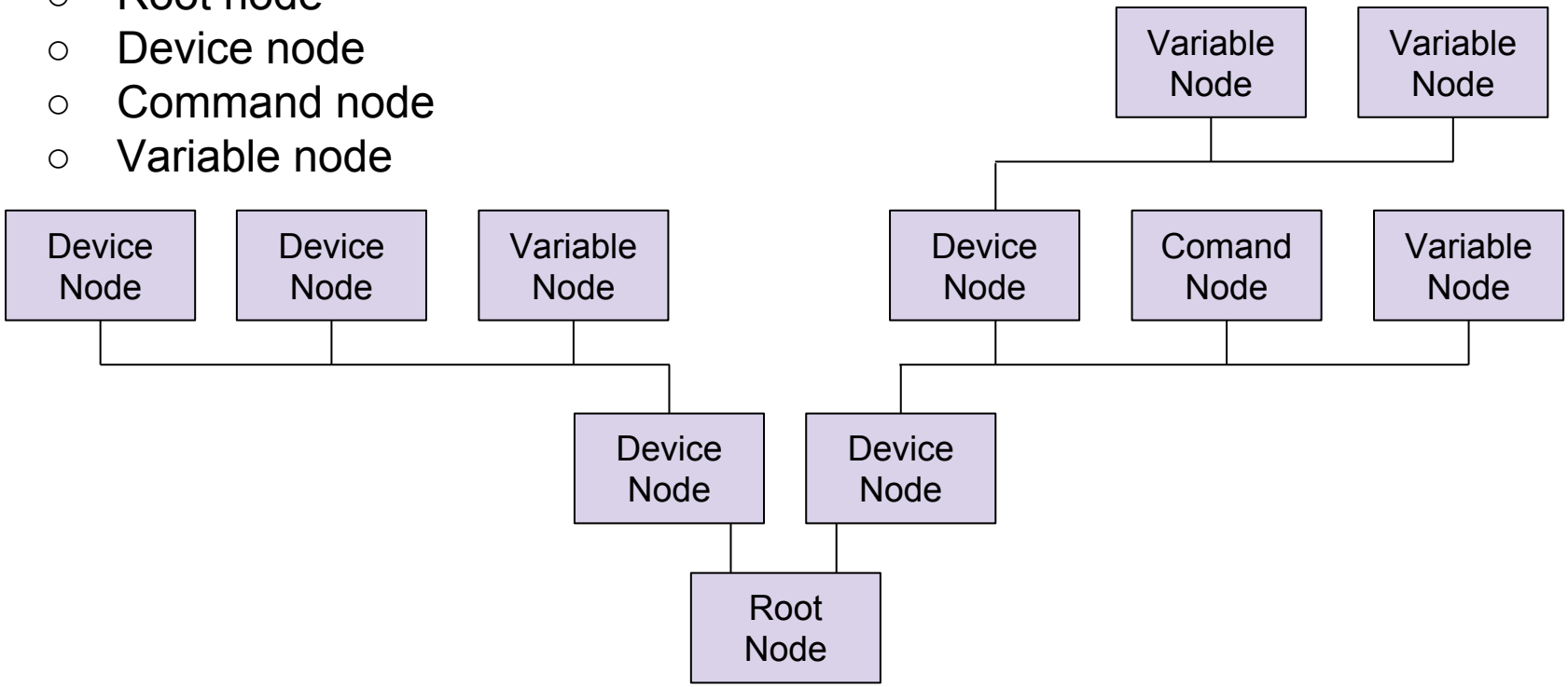
Rogue Example




```
evalBoard = pyrogue.Root('evalBoard','Evaluation Board')
evalBoard.add(pyrogue.RunControl(rates={1:'1 Hz', 10:'10 Hz',30:'30 Hz'}))

dataWriter = pyrogue.utilities.fileio.StreamWriter('dataWriter')
evalBoard.add(dataWriter)

pgpVc0 = rogue.hardware.pgp.PgpCard('/dev/pgpcard_0',0,0)
pgpVc1 = rogue.hardware.pgp.PgpCard('/dev/pgpcard_0',0,1)
pgpVc2 = rogue.hardware.pgp.PgpCard('/dev/pgpcard_0',0,2)

srp = rogue.protocols.srp.SrpV0()
pyrogue.streamConnectBiDir(pgpVc0,srp)

pyrogue.streamConnect(evalBoard,dataWriter.getChannel(0x0))
pyrogue.streamConnect(pgpVc1,dataWriter.getChannel(0x1))
pyrogue.streamConnect(pgpVc2,dataWriter.getChannel(0x2))

prbsRx = pyrogue.utilities.prbs.PrbsRx('prbsRx')
pyrogue.streamTap(pgpVc1,prbsRx)
evalBoard.add(prbsRx)

evalBoard.add(surf.axi.AxiVersion(memBase=srp,
                                  offset=0x0))

evalBoard.add(surf.protocols.ssi.SsiPrbsTx(memBase=srp,
                                           offset=0x30000))

evalBoard.start(pyroGroup='rogueTest')

epics = pyrogue.epics.EpicsCaServer(base='rogueTest',root=evalBoard)
epics.start()
```

PyRogue Management Tree

- PyRogue provides a mechanism for organizing elements into a tree
  - The tree structure is arbitrary and does not necessarily follow the hardware bus structures
  - Elements within the tree are exposed to outside systems via the various management interfaces (epics, pyro, mysql, etc)
  - Tree structure is implemented fully in Python
- Node types
  - Root node
  - Device node
  - Command node
  - Variable node



PyRogue Node

SLAC

- A Node is the base building block of pyrogue elements contained in a tree

- Each node has the following attributes:
  - name: Node name which is unique at its level in the tree
  - description: Description of node, usually a helpful note to the user
  - hidden: True if node is hidden from external management entities
  - path: Full path to element in tree in form a.b.c.d
  - expand: Node should be expanded in GUI by default

- Sub-nodes are available as attributes of the node:
  - node.subnode.variable.name
  - node.subnode.variableArray[3].name

Sub Node Access

- The following node access methods are provided
  - nodes(): Return a dictionary {name:node} of the child nodes
  - variables(): Return a dictionary {name:node} of the child Variables
  - commands(): Return a dictionary {name:node} of the child Commands
  - devices(): Return a dictionary {name:node} of the child Devices
  - parent(): Return a reference to the parent node
  - root(): Return a reference to the root node
  - node(name): Return a node with the provided name

- The following node add methods are provided

  - add(node): Add a child node

  - addNodes(type,number,stride,args): Add an array of nodes
    - Offset of each node is adjusted using the stride arg
    - args is passed to the creator of the node of the passed type
    - The passed name value is adjusted to indicate an array: name[n]

Sub Node Access

SLAC

- A find method is availabe in the Node which allows one to find sub-nodes with a power set of matching filters
  - Find can be initiated at any level in the tree to find sub-nodes

- find(recurse=True, typ=None, **kwargs)
  - recurse: Search beyond this level
  - If typ is not None, filter on nodes of a certain type
  - The rest of the args are a property/value combinations
    - Regular expressions are used if the value is a string

find(recurse=True, type=pyrogue.RemoteVariable, name="*Pad*, mode='RW')

Find all nodes of type RemoteVariable with a name containing 'Pad' with mode = 'RW'

PyRogue Variables

SLAC

- A Variable node is the management interface for updating and monitoring values within the system
  - BaseVariable - abstract base class for the other 3
  - LocalVariable - a local python variable
  - RemoteVariable - A remote register in hardware
  - LinkVariable - Allows one or more Variables to be viewed or controlled together through a custom translation function

- All variable types have the following BaseVariable attributes (in addition to the Node attributes)
  - mode: What types of accesses are supported by this variable (‘RW’,’RO’,’WO’)
  - value: An optional default value for the variable
  - update: True/False indication enabling update notifications to variable listeners
  - disp: A display string indicating how the variable value should be displayed and parsed
  - enum: A dictionary of key,value pairs for variables which have a set of selections
  - revEnum: A reversed representation of the enum dictionary (value as key)
  - units: Optional string field indicating the units of the variable for display purposes
  - minimum: Optional minimum value for a variable with a set range
  - maximum: Optional maximum value for a variable with a set range
  - pollInterval: The period at which this variable should be polled (0=disable polling)
  - dependencies: A list of variable which this variable is dependent on (Linked variable)
  - typeStr: String indicating the variable type: ‘Uint(32)’, ‘String’, ‘str’, ‘int’
  - nativeType: Native type of variable
    - Same as type(self.value())

Variable Data Access

- Variables can either be accessed with a raw type (low level integer in a register) or a display type (string representation of the variable, or enum value)
  - Example: enum = {4:'four', 5:'five'}
    - Raw value = 4, display value = 'four'
  - Example: value = 32, disp='{:#x}', display value = '0x20'

- get(read=True): Return the raw value of the variable
  - If read=True first read the hardware before returning value
  - If read=False just return the shadow value

- value(): Equivalent to get(read=False)

- genDisp(value): Helper method to convert the passed raw value into the display representation
  - genDisp(0x32) will return '0x20' when disp='{:#x}'

- getDisp(read=True): Return the display value with optional read
  - Equivalent to genDisp(get(read))

- valueDisp(): Return the display value without a read
  - Equivalent to genDisp(get(read=False))

Variable Data Access

- set(value,write=True): Update the raw value of the variable.
  - If write=True a blocking transaction to hardware is generated immediately
  - If write=False the shadow value is updated in anticipation of a future write during internal bulk operations (config from file as an example)
- post(value): Update the raw value of the variable and generate an immediate non blocking write
- parseDisp(sValue): Helper method to convert a string representation or enum value into a raw value
  - parseDisp('0x20') will return 32 when disp = '{:#x}'
  - parseDisp will detect when it is being passed the raw value: parseDisp(32) will return 32
- setDisp(sValue,write=True): Update the variable with the display value with optional write
  - Equivalent to: set(parseDisp(sValue),write)

- ~~Variables can be accessed like any other python variable~~
  - ~~variableA = variableB + 1 is the same as variableA.set(variableB.get()+1)~~

- LocalVariables can contain complex data types such as lists and dictionaries
  - typeStr and nativeType will return the container type
  - setDisp, getDisp, valueDisp will not work properly

Variable Update Notification

SLAC

- Variable can store a list of methods and variables which are used to notify other elements when a variable's values has changed
  - As a result of a set() or setDisp()
  - As a result of a recent read from hardware
- updated() method notifies the variable when new data is available
- addListener() method can be used to add callbacks
  - addListener(variable) will generate an update notification to the passed variable's linkUpdated() method which in turn updates its listeners (see LinkedVariable description)
  - addListener(function) will register a callback
- The listener callback is a method with the following signature:
  - callback(var,value,disp)
    - var is a reference to the variable which generated the update
    - value is the raw value following the update
    - disp is the display value following the update
- External object can also subscribe to variable update notifications at the tree level
  - Only called when update=True
  - root.addVarListener(function)
  - Where function is: function(yaml, list)
    - yaml is a yaml representation of the variables that were updated
      - Same format as configuration file
    - List is a list of the variables that were updated with each entry having the format:
      - {'path': variable_path, 'value': raw_value, 'disp': display_value}

LocalVariable

- A LocalVariable is used to manage a local value such as a filename, software mode, etc
  - Can be linked to localSet and localGet functions which are called during access

- var = LocalVariable(name, description='', update=True, mode='RW', value=None, disp='{}', enum=None, units=None, hidden=False, minimum=None, maximum=None, localSet=None, localGet=None, pollInterval=0)
  - Value must be passed when creating a local variable.
    - Used to set a default value as well as determine the native type

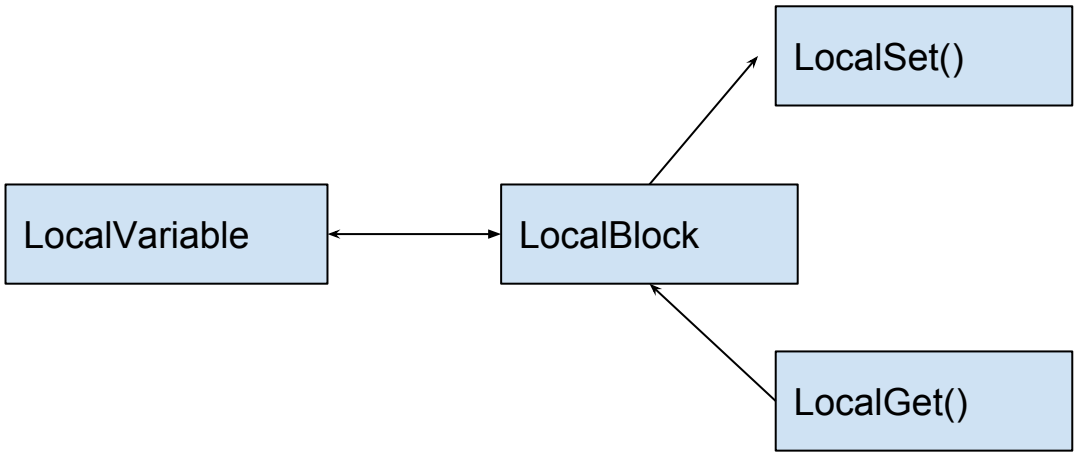
- localSet and localGet are references to functions which are used for set and get calls
  - Optional args. Passed only when get and set need to do something special.
  - Called within the context of a lock associated with the variable
  - Have a set of supported args which are optional and detected by the variable
    - Arg names must match the template to be matched
- C++ functions exposed through python are supported but no args are passed

- localSet is a function which is called when the variable's set() method is called
  - Called after the internal shadow value is updated
  - Template: localSet(dev, var, value, changed)
    - dev = reference to device containing the variable
    - var = reference to the variable generating the call
    - value = updated value after set
    - changed = boolean indicated the value has changed as a result of the set

LocalVariable

- localGet is a function which is called when the variable's get() method is called
  - Expected to return a new value which will be stored in the shadow value
  - Template: localGet(dev, var)
    - dev = reference to device containing the variable
    - var = reference to the variable generating the call

- Each LocalVariable is backed by a LocalBlock which holds the shadow value
  - The LocalBlock executes localSet and localGet



LocalVariable Example From Prbs Receiver

SLAC

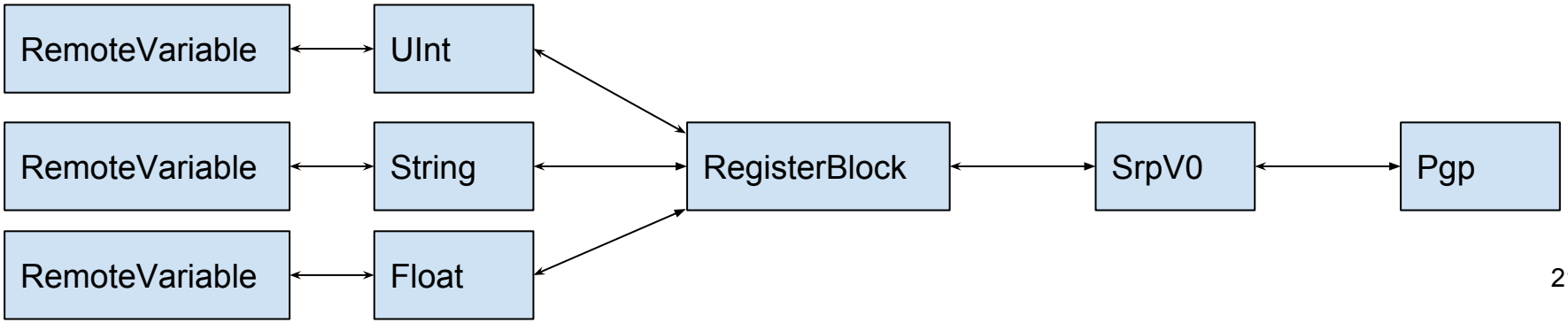
```
self._prbs = rogue.utilities.Prbs()

self.add(pyrogue.LocalVariable(name='rxErrors', description='RX Error Count',
                               base='uint', mode='RO', pollInterval=1, value=0,
                               localGet=self._prbs.getRxErrors))

self.add(pyrogue.LocalVariable(name='rxCount', description='RX Count',
                               base='uint', mode='RO', pollInterval=1, value=0,
                               localGet=self._prbs.getRxCount))

self.add(pyrogue.LocalVariable(name='rxBytes', description='RX Bytes',
                               base='uint', mode='RO', pollInterval=1, value=0,
                               localGet=self._prbs.getRxBytes))
```

RemoteVariable

- A RemoteVariable is used to manage a remote value typically located on the managed hardware
  - Associated with an address, offset and size
  - Uses a ‘Base’ class to convert between raw value and hardware bits

- var = RemoteVariable(name, description=’’, update=True, mode=’RW’, value=None, disp=’{}’, enum=None, units=None, hidden=False, minimum=None, maximum=None, base=pyrogue.UInt, offset=None, bitSize=32, bitOffset=0, pollInterval=0, verify=True)
  - base: Pointer to a Model class used to convert between the variable raw value and register bits
  - offset: Defines the offset in bytes of the variable relative to the device
  - bitOffset: Defines the offset in bits of the LSB relative to the offset in bytes
    - Offset = 0, bitOffset=32 is equivalent to offset=4, bitOffset=0
  - bitSize: Number of bits occupied by the variable
  - verify: True/False flag indicating if a verify should occur for a RW variable following a write
  - value: Optional to set a default value (base is used to determine type)

- A RegisterBlock serves as the gateway between the variable and a memory interface device
  - RemoteVariables are associated into common RegisterBlocks by detecting overlaps in accessed space



RemoteVariable - Advanced Bit Mappings

- The typical use for offset, bitOffset and bitSize involves passing a single integer value to each to indicate the byte and bit offsets as well as the size
  - In some cases the variable value will be mapped to bits scattered around memory space

- Rogue supports the ability to provide a list in any or all of the three attributes
  - If any of the three values are a list, the others must match in length, have a length of 1 or be an integer value
    - Values that are integers or a list with a length of 1 will be replicated to match the longer list

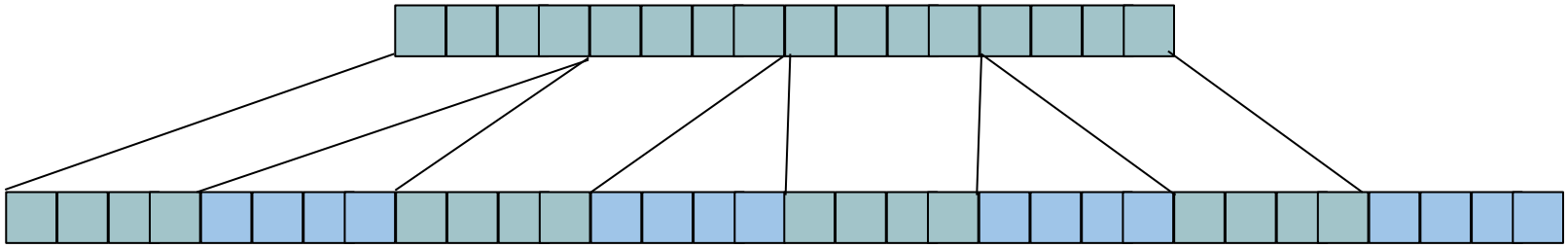
Offset = 100, bitOffset = [0,8,16,24], bitSize = [4,4]

will generate an error

Offset = 100, bitOffset = [0,8,16,24], bitSize = [4]

will be converted to

Offset = [100,100,100,100], bitOffset = [0,8,16,24], bitSize = [4,4,4,4]



RemoteVariable Examples From AxiVersion

SLAC

```
self.add(pr.RemoteVariable(
    name         = "UpTimeCnt",
    description  = "Number of seconds since last reset",
    offset       =  0x08,
    bitSize      =  32,
    bitOffset    =  0x00,
    base         = pr.UInt,
    mode         = "RO",
    pollInterval  = 1
))

self.add(pr.RemoteVariable(
    name         = "BuildStamp",
    description  = "Firmware Build String",
    offset       =  0x800,
    bitSize      =  8*256,
    bitOffset    =  0x00,
    base         = pr.String,
    mode         = "RO",
))
```

Model Class

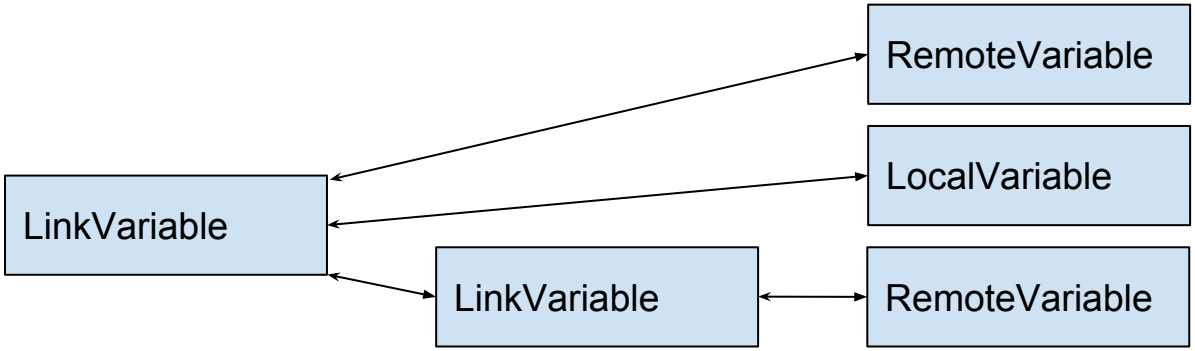
- RemoteVariable takes a **base=** argument in its constructor
- This describes the low level data type
- Several classes are defined in that can be passed to **base=**
  - pyrogue.UInt - Unsigned Integer (default if base= is not passed)
  - pyrogue.Int - Signed Integer
  - pyrogue.Bool - A 1-bit Boolean
  - pyrogue.String - A null terminated ASCII byte string (C style)
  - pyrogue.Float - A floating point number
- These classes define how a Python native variable gets converted into bits/bytes written to hardware.
- The RemoteVariable **bitSize=** argument determines the size for the conversion.
- All of the current classes only do Little-Endian conversion. We can add Big-Endian variants if needed.
- Each Base class also defines a default **disp=** that RemoteVariable will use unless you specify your own
  - pyrogue.UInt - '{:#x}' - Hexidecimal - 0x1234
  - pyrogue.Int - '{:d}' - Decimal
  - pyrogue.Bool - {False: 'False', True: 'True'} - Enum
  - pyrogue.String - '{}' - Already a string
  - pyrogue.Float - '{:f}' - Default float format

LinkVariable

- A LinkVariable is a virtual variable which does not hold any local value
  - Passes values through with possible local modification using linkedSet and linkedGet functions
  - Dependent on other variables

- var = LinkVariable(name, description='', mode='RW', disp='{}', enum=None, units=None, hidden=False, minimum=None, maximum=None, typeStr='Linked', linkedGet=None, linkedSet=None, dependencies=None)
  - typeStr: Set the typeStr attribute
  - dependencies: List of variables which this variable is dependent on
  - value is not used

- linkedSet and linkedGet are references to functions which are used for set and get calls
  - Have a set of supported args which are optional and detected by the variable
    - Arg names must match the template to be matched
- C++ functions exposed through python are supported but no args are passed



LinkVariable

SLAC

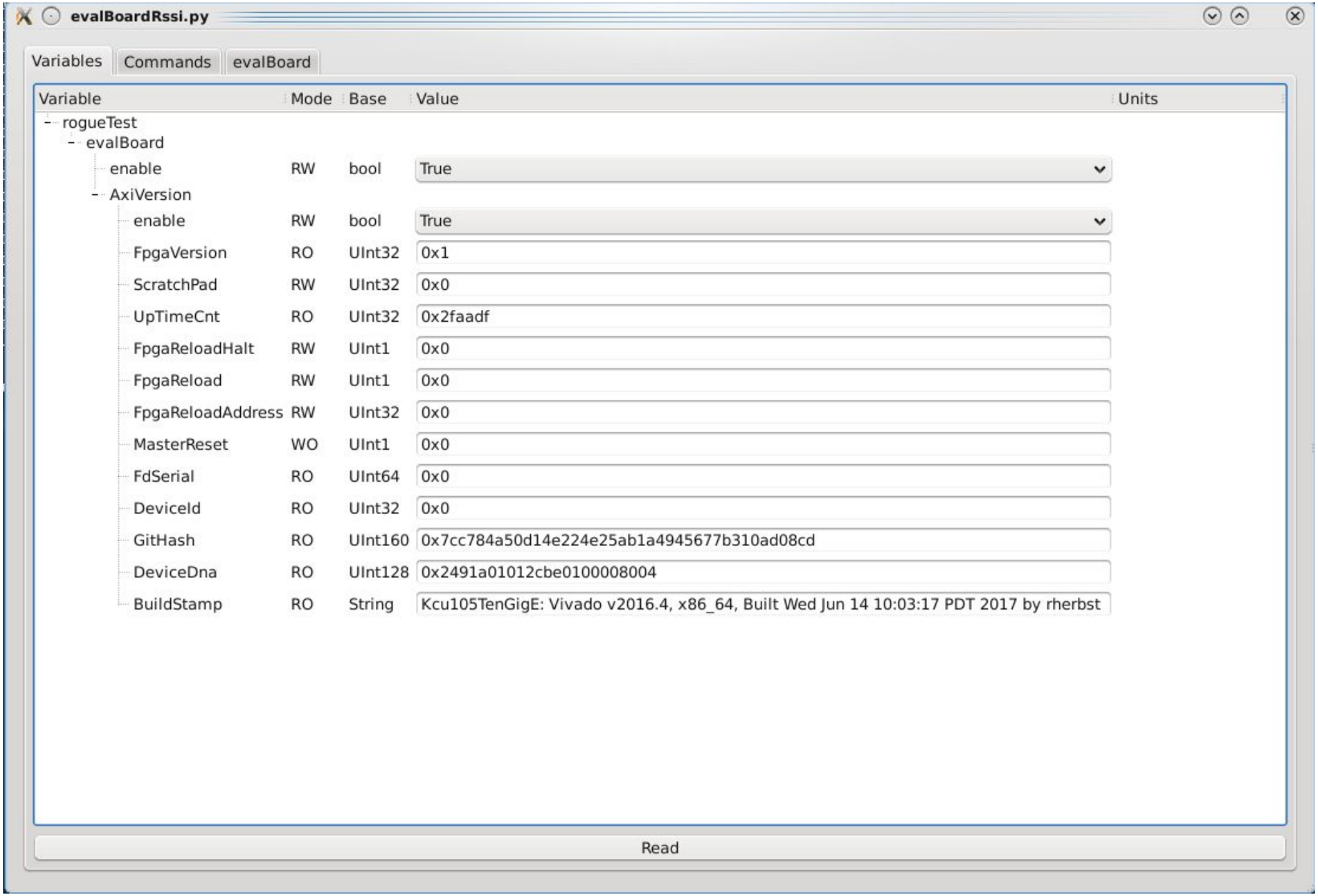
- linkedGet(dev,var,read)
  - dev: reference to the device containing this variable
  - var: reference to the variable
  - read: read arg passed to get() (should be passed through to linked variables)
  - Returns a value which is returned by get()

- linkedGet(dev,var,value,write)
  - dev: reference to the device containing this variable
  - var: reference to the variable
  - value: raw value passed to set()
  - write: write arg passed to set() (should be passed through to linked variables)

- Example:

```
var1 = pr.RemoteVariable(name='OT_LimitRaw', offset=0x4000, bitSize=12, bitOffset=4, hidden=True)

var2 = pr.LinkVariable(name='OT_Limit', units='degC',
                       linkedGet=getTemp, linkedSet=setTemp, dependencies=[var1]))

def getTemp(var,read):
    value = var.dependencies[0].get(read)
    return (float(value)*(503.975/4096.0)) - 273.15

def setTemp(var, value, write):
    ivalue = int((value + 273.15)*(4096/503.975))
    var.dependencies[0].set(ivalue,write)
```

Variables


evalBoardRssi.py
Variables
Commands
evalBoard
Variable
Mode
Base
Value
Units
rogueTest
evalBoard
enable RW bool True
AxiVersion
enable RW bool True
FpgaVersion RO UInt32 0x1
ScratchPad RW UInt32 0x0
UpTimeCnt RO UInt32 0x2faadf
FpgaReloadHalt RW UInt1 0x0
FpgaReload RW UInt1 0x0
FpgaReloadAddress RW UInt32 0x0
MasterReset WO UInt1 0x0
FdSerial RO UInt64 0x0
DeviceId RO UInt32 0x0
GitHash RO UInt160 0x7cc784a50d14e224e25ab1a4945677b310ad08cd
DeviceDna RO UInt128 0x2491a01012cbe0100008004
BuildStamp RO String Kcu105TenGigE: Vivado v2016.4, x86_64, Built Wed Jun 14 10:03:17 PDT 2017 by rherbst
Read

PyRogue Commands

SLAC

- A Command node is the management interface for executing commands within the system
    - BaseCommand - abstract base class for the other Command classes
    - LocalCommand
    - RemoteCommand
  - All commands are sub-classes of BaseVariable and support all variable attributes
    - RemoteCommand is also a sub-class of Remote Variable

- All command types have the following BaseCommand attributes (in addition to the Node attributes)
  - value: An optional value to determine the arg type
  - disp: A display string indicating how the command arg should be parsed
  - enum: A dictionary of key,value pairs for args which have a set of selections
  - revEnum: A reversed representation of the enum dictionary (value as key)
  - minimum: Optional minimum value for a arg with a set range
  - maximum: Optional maximum value for a arg with a set range
  - arg: True if an argument is supported

- A command has a function which is called when the command is executed
  - Has a set of supported args which are optional and detected by the variable
    - Names must match the template to be matched
- C++ functions exposed through python are supported but no args are passed

- function(dev, cmd, arg)
    - dev = reference to device containing the variable
    - cmd = reference to the command generating the call
    - arg = arg passed by the command execution
      - The presence of arg determines is an arg is supported by the command

LocalCommand & Remote Command

SLAC

- A LocalCommand is a command which does not have a 1:1 associated with a hardware element
  - Used to execute a sequence instructions within software
  - Can include sequences of variable writes or other command executions

- cmd = LocalCommand(name, description='', value=0, disp='{}', enum=None, minimum=None, maximum=None, function=None)
  - Value determines the command arg type

- A RemoteCommand is a command which has a 1:1 associated with a hardware element
  - Subclass of RemoteVariable
  - Used to generate a set of writes to the associated hardware element

- cmd = RemoteCommand(name, description='', value=None, disp='{}', enum=None, minimum=None, maximum=None, base=pyrogue.UInt, offset=None, bitSize=32, bitOffset=0, function=None)
  - Passed attributes are the same as for BaseCommand and RemoteVariable
  - Value is optionally used to set an initial value in the underlying Block
  - Base is used to determine the arg type
  - Several common functions provided (see next page)

- Commands can either be executed directly or with the call() method:
  - command(arg)
  - command.call(arg)

Pre-canned Command Functions

SLAC

- A number of commonly used commands and sequence generators are provided to support RemoteCommand operations

- True functions. Passed as a reference: function=BaseCommand.nothing

  BaseCommand.nothing():
  Do nothing
  BaseCommand.toggle():
  set() the underlying variable to 1 then 0
  BaseCommand.touch():
  Execute a single set(1) if an arg is not passed, otherwise call set with the passed arg
  BaseCommand.touchZero():
  Execute a single set(0)
  BaseCommand.touchOne():
  Execute a single set(1)
  BaseCommand.postedTouch():
  Execute a single post(1) if an arg is not passed, otherwise call post with the passed arg

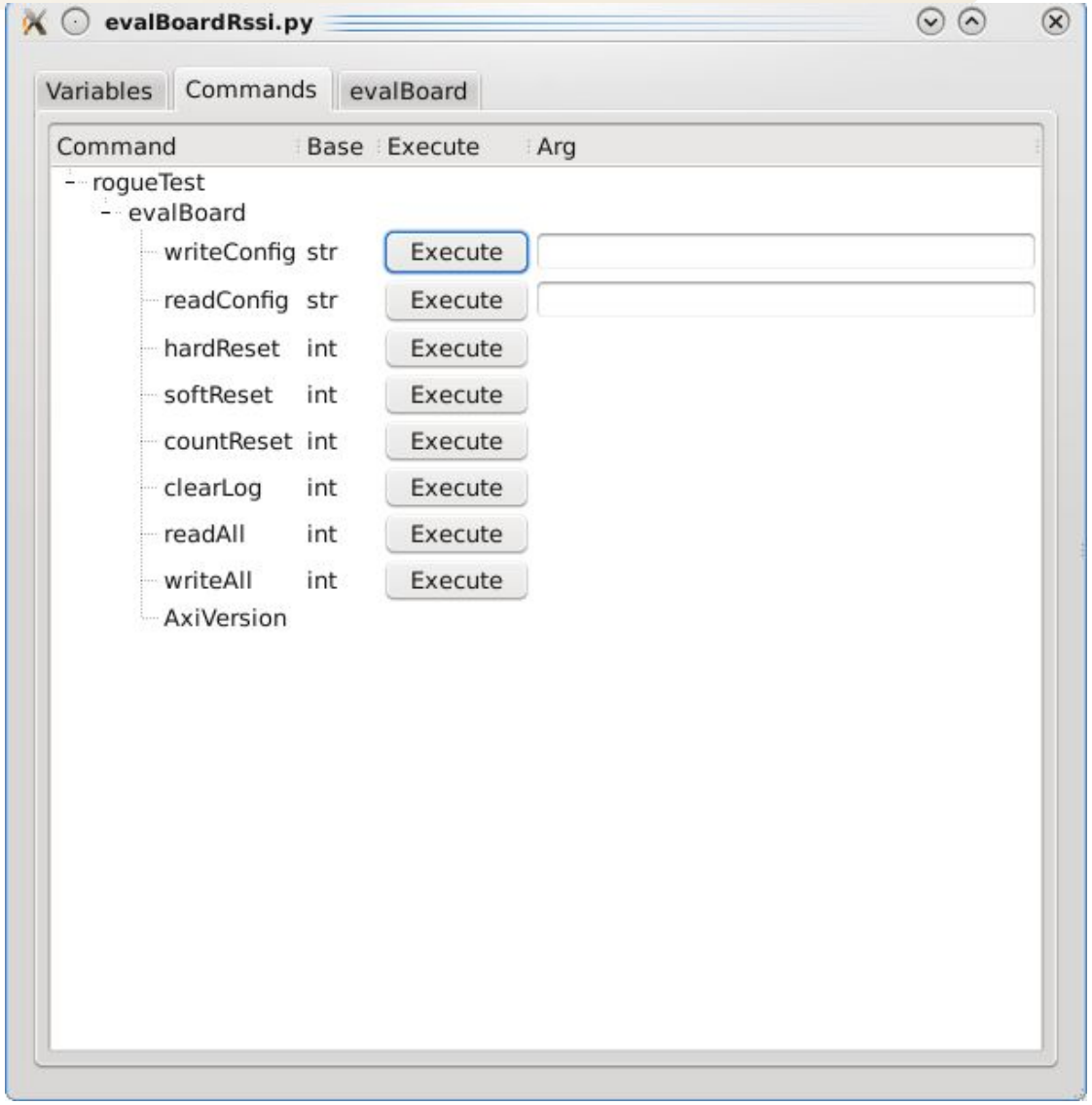
- Function creators. Called to return a function: function=BaseCommand.createToggle([1,2,3])

  BaseCommand.createToggle(sets):
  Create a function generating a set() call for each value in [sets]
  BaseCommand.createTouch(value):
  Create a function generating a single set() called with passed value

Remote Command Examples

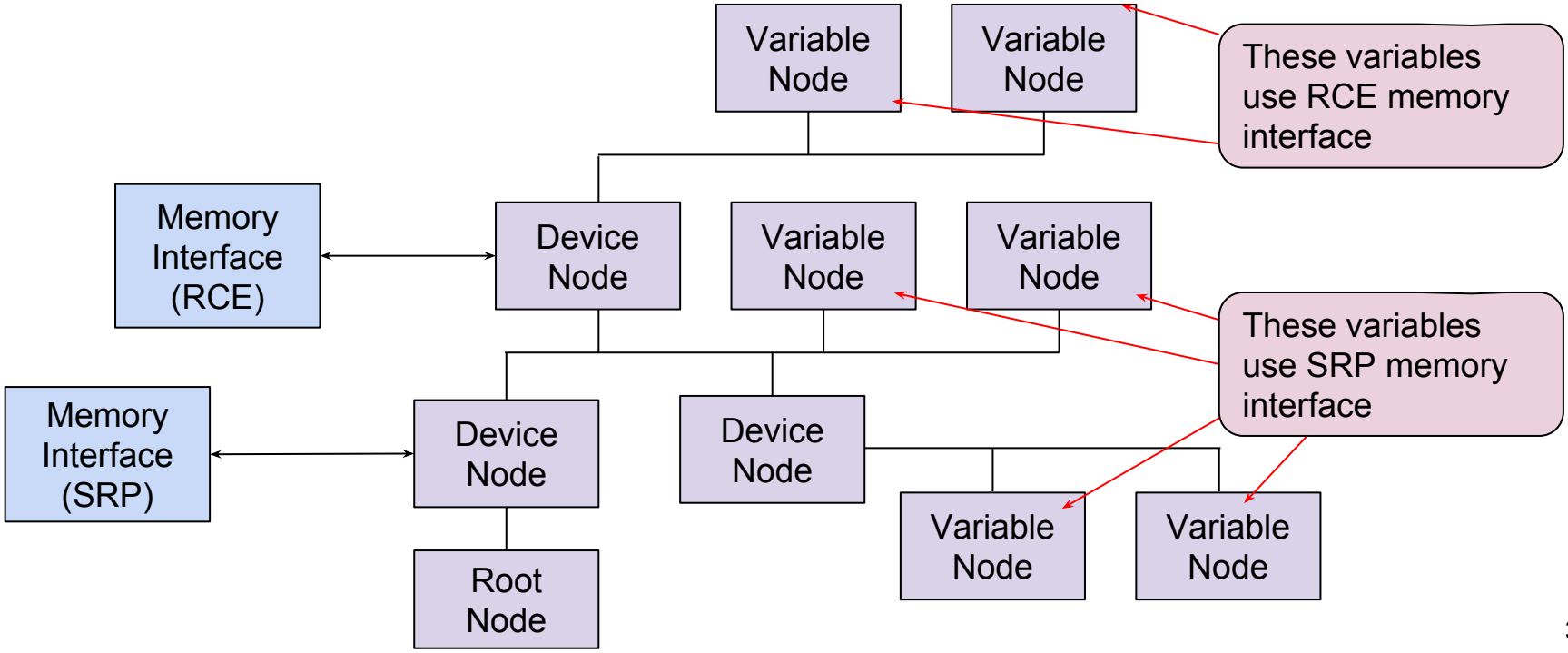
```
self.add(pr.RemoteCommand(
    name        = 'CountReset',
    offset      = 0x00,
    bitSize     = 1,
    bitOffset   = 0,
    function    = pr.BaseCommand.toggle,
))

self.add(pr.RemoteCommand(
    name        = "ResetRx",
    offset      = 0x04,
    bitSize     = 1,
    bitOffset   = 0,
    function    = pr.BaseCommand.toggle,
))
```

Commands


evalBoardRssi.py
Variables
Commands
evalBoard
Command
Base
Execute
Arg
rogueTest
evalBoard
writeConfig str Execute
readConfig str Execute
hardReset int Execute
softReset int Execute
countReset int Execute
clearLog int Execute
readAll int Execute
writeAll int Execute
AxiVersion

PyRogue Device

- A pyrogue.Device node is a container for Variable and Commands as well as other devices
- A Device class is a sub-class of rogue::interfaces::memory::Hub
  - This allows it to serve as a memory master as well as act as a hub to other memory masters
  - Variables (and their connected Blocks) associated with hardware will have an offset relative to the Device base address and will use the Device's linked memory::Slave when performing register accesses
  - Added Devices which are not already associated with a memory Slave interface will inherit the base Device's base address and Slave interface
  - A Device can have it's own direct link to a memory Slave which is not related to parent.



Device

- A Device has the following attributes (in addition to those inherited from Node):
  - offset: Relative address offset in bytes.
  - address: Full system level address after applying the offsets of full path to a memory master
  - size: Optional field for Devices which utilize _rawWrite and _rawRead methods
  - memBase
    - Object which provides a path to a memory Slave device. Can either be an instance of a memory Slave or an instance of a memory Hub from which it will inherit a base address and Slave instance link
    - If memBase is not provided the memBase will be inherited when the device is added to a parent Device

- Each device also has a special Variable named ‘enabled’. This variable is an on/off switch for the device, its local variables as well as all sub-devices.
  - Allows some portions of the tree to be turned off
  - Sub-devices which are disabled due to a parent device being disabled will show as ‘ParentFalse’ to differentiate from a local value of ‘False’

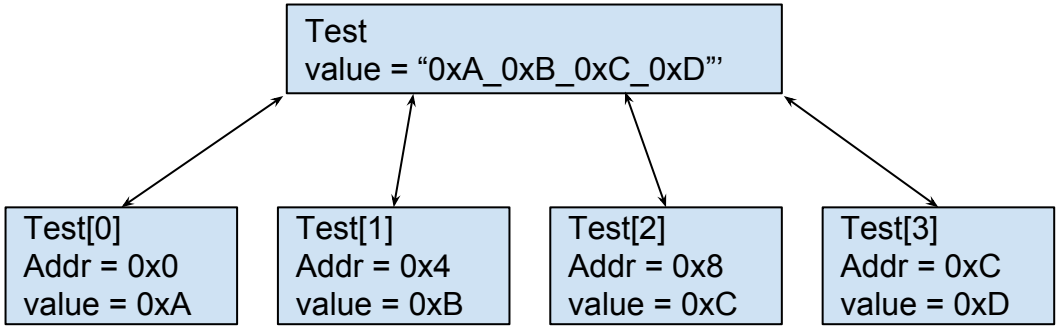
- dev = Device(name=None, description=’’, memBase=None, offset=0, size=0, hidden=False, expand=True, enabled=True)
  - enabled: Is the initial state of the ‘enabled’ variable
  - size: A size value of non-zero should be provided for empty devices which will make use of the _rawWrite and _rawRead calls.

Adding RemoteVariable Arrays In Device

- Sub-Devices, Varibles and Commands are added using the add() method

- A special device-specific add method allows the creation of RemoteVariable arrays
  - addRemoteVariables(number, stride, pack=False, **kwargs)
    - This will create an array of RemoteVariables containing ‘number’ elements with memory address offset stride of ‘stride’ bytes
    - The provided name will be adjusted to indicate an array: name[n]
    - Works in a similar fashion to addNodes() at the node level if pack=False
    - When pack=True an additional LinkVariable will be created which is linked to the array of RemoteVariables and the RemoteVariables will be hidden
      - This LinkVariable will represent the data in the RemoteVariable array as a string containing the ‘display’ values of each variable
        ‘0x1a_0x1b_0x1c_0x1d’

- Example:
  addRemoteVariables(number = 4, stride=4, offset=0, bitSize=4, name=’test’,base=UInt, disp=’{:#x}’)



Device Methods

- Devices have a number of supported methods, some can be overridden by a sub-class
- hideVariables(hidden, variables=None)
  - This method allows all of the variables associated with a device to be hidden or unhidden
    - hidden: Is the True/False value to set each variable to
    - variables: is a list of variable objects or variable names to apply the hidden arg to
- hardReset(self)
  - This method is called on each Device when a system wide hardReset is generated
  - Can be overridden by a sub class to issue a set of commands or variable sets
- softReset(self)
  - This method is called on each Device when a system wide softReset is generated
  - Can be overridden by a sub class to issue a set of commands or variable sets
- countReset(self)
  - This method is called on each Device when a system wide countReset is generated
  - Can be overridden by a sub class to issue a set of commands or variable sets
- Note that the exact meaning of hardReset, softReset and countReset may be experiment specific.

Device Read/Write Operations

SLAC

- When system wide config, write and read operations are generated the blocks in each Device are manipulated by the top level Root node
  - In a bulk config the shadow values in each variable are written by issuing the setDisp() call of each variable with the value contained in the configuration file
    - A write() transaction in each block is then generated and added to the queue
    - A verify() transaction in each block is then generated and added to the queue
    - The system then waits for the previous two transactions in each block to complete and then checks the result. Updated variables are then notified.
  - In a bulk read operation:
    - A read() transaction in each block is then generated and added to the queue
    - The system then waits for the read transaction in each block to complete and then checks the result.

- Similarly individual variable writes (from epcis, a GUI or a script) will result in transactions occurring on each Block
  - A variable set(write=True) issues the following sequence:
    - set() the value to the variable
    - Issue a write to the associated Block
    - Issue a verify to the associated Block
    - Check the result of the transactions and generate updates.
  - A variable get(read=True) issues the following sequence:
    - Issue a read from the associated block
    - Check the result of the transaction
    - Return the updated value and generate updates.

Custom Read/Write Operations In A Device

SLAC

- In some special Devices this above order may not work. Some Devices may require a special command sequence to be issued before and/or after each write or read transaction.
- The Device class allows the user to override the Device level functions that are called during these read and write operations

- writeBlocks(self, force=False, recurse=True, variable=None)
  - This method is called to issue a write transaction to the blocks within a Device
  - The following args are passed:
    - force: This flag indicates if the transaction will force writes for blocks which are not stale
    - recurse: This flag indicates if the writeBlocks operation should be forwarded to the sub-devices
    - variable: This is a flag which will reference a variable for single variable transactions

Example custom writeBlocks:

```
def writeBlocks(force=False, recurse=True, variable=None):

    // Do some stuff before the transactions

    super(writeBlocks, self)(force=force,recurse=recurse,variable=variable)

    // Do some stuff after the transactions
```

Custom Read/Write Operations In A Device

SLAC

- verifyBlocks(self, recurse=True, variable=None)
  - This method is called to issue a verify transaction to the blocks within a Device
  - The following args are passed:
    - recurse: This flag indicates if the verifyBlocks operation should be forwarded to the sub-devices
    - variable: This is a flag which will reference a variable for single variable transactions
- readBlocks(self, recurse=True, variable=None)
  - This method is called to issue a read transaction to the blocks within a Device
  - The following args are passed:
    - recurse: This flag indicates if the readBlocks operation should be forwarded to the sub-devices
    - variable: This is a flag which will reference a variable for single variable transactions

Example custom verifyBlocks and readBlocks:

```
def verifyBlocks(recurse=True, variable=None):
    // Do some stuff before the transactions
    super(verifyBlocks, self)(recurse=recurse,variable=variable)
    // Do some stuff after the transactions

def readBlocks(recurse=True, variable=None):
    // Do some stuff before the transactions
    super(readBlocks, self)(recurse=recurs,variable=variable)
    // Do some stuff after the transactions
```

Read/Write Operations Within A Device

SLAC

- The last overridable method is called to wait on the result of a set of bulk transactions or a single set() or get transaction()
  - The overridden writeBlocks() or readBlocks() can call this to force any queued transactions to complete before issuing more transactions
    - Set an enable bit
    - Start a processing thread

- checkBlocks(self, recurse=True, variable=None)
  - This method is called to issue a transaction check to the blocks within a Device
  - The following args are passed:
    - recurse: This flag indicates if the checkBlocks operation should be forwarded to the sub-devices
    - variable: This is a flag which will reference a variable for single variable transactions

Example custom checkBlocks:

```
def checkBlocks(recurse=True, variable=None):

    super(verifyBlocks, self)(recurse=recurse,variable=variable)
    // Do some stuff after the checks are done
```

Command Decorators

SLAC

- Two types of command decorators can be used in Device to create LocalCommands
  - Decorators on class methods
  - Decorators on functions created within init()

- Class method decorators:
  - The following code is from the Root class which issues a readConfig from a file:

```
@pyrogue.command(order=1, name='readConfig', value='', description='Read config')
def _readConfig(self,arg):
```

    - This decorator will create a LocalCommand named ‘readConfig’ and pass the _readConfig method as the function
      - If name was not provided the LocalCommand would use the method name
    - Any decorator args are passed through to the LocalCommand creation
    - Because the method has the argument ‘arg’, the LocalCommand will be configured to accept arguments. If it was not there arguments would be disabled.
      - value=’’ sets the arg type as string
    - The order arg in the decorator is used to control the order of the LocalCommand creation
    - The decorated method has the same signature as the functions described in the LocalCommand section of this presentation

```
@pyrogue.command(order=6, name="readAll", description='Read all values')
def _read(self):
```

    - This command does not have an arg and value is not passed because it is not necessary

Command Decorators

- __init__() decorators:
  - The following code is from the Lmk04828 class:

```
def __init__(...):
    ...
    @self.command(description="Load the CodeLoader .MAC file",value='',)
    def LoadCodeLoaderMacFile(arg):
        ...
```

    - This decorator will create a LocalCommand named ‘LocalCodeLoaderMacFile’ and pass the decorated method as the function
      - Alternative a name arg could be passed to the decorator to override the name
    - Any decorator args are passed through to the LocalCommand creation
    - Because the command has the argument arg, the LocalCommand will be configured to accept arguments. If it was not there arguments would be disabled.
      - value=’’ sets the arg type as string
    - The decorated method has the same signature as the functions described in the LocalCommand section of this presentation

- Currently decorators can only be used to create LocalCommands
  - This may be expanded to include RemoteCommands as well
  - Can be detected as RemoteCommands when offset is passed in the decorator

Raw Read & Write Calls Within A Device

SLAC

- In some cases the user will want to create a Device which accesses memory directly without exposing values through Variables
  - Prom programming or reading DAC waveforms from a file
- A Device can be created with an offset and size and access memory within its mapped space using _rawWrite and _rawRead calls
  - Operations are usually triggered by a LocalCommand or LocalVariable within the device

- def _rawWrite(self, offset, data, base=pyrogue.UInt, stride=4, wordBitSize=32):
  - offset : Offset in bytes from Device base address
  - data: data to write in the form of a single value or a list of values
  - base: base class indicating the type of value(s) being written
  - stride: the spacing between the value locations in bytes
  - wordBitSize: the size of each value in bits

- def _rawRead(self, offset, numWords=1, base=pyrogue.UInt, stride=4, wordBitSize=32, data=None):
  - offset : Offset in bytes from Device base address
  - numWords: Number of values to read (if data=None)
  - base: base class indicating the type of value(s) being read
  - stride: the spacing between the value locations in bytes
  - wordBitSize: the size of each value in bits
  - data: optional list in which the read data values will be stored
    - If data list is provided the number of values read is determined by the list length

- _rawWrite and _rawRead are blocking calls

Device Examples

- Custom Devices are created by sub-classing pyrogue.Device

- Open up this link: https://github.com/slaclab/rogue/blob/master/python/pyrogue/utilities/prbs.py
  - Scroll down to PrbsRx
    - This is a Device class which interfaces to an underlying c++ class which receives and checks prbs data
    - Note the creation of LocalVariables which call the underlying c++ class method directly
    - Note the countReset() method
  - Scroll down to PrbsTx
    - Note the genFrame LocalCommand which generates a call to the underlying c++ class
      - It also gets the value of a pure local variable (txSize) which is passed as an arg to the c++ class
    - Note the txEnable variable which uses the changed arg to determine whether to start or stop a backup thread in the C++ class using enable and disable

- AxiVersion: https://github.com/slaclab/surf/blob/cryo-det-dev/python/surf/axi/_AxiVersion.py

- SsiPrbsTx: https://github.com/slaclab/surf/blob/cryo-det-dev/python/surf/protocols/ssi/_SsiPrbsTx.py
  - Note the command decorator which issues a write to an un-associated variable

- Lmk0428: https://github.com/slaclab/surf/blob/cryo-det-dev/python/surf/devices/ti/_Lmk04828.py

PyRogue Root Node

SLAC

- The root node is the base of the tree and provides tree wide access methods
  - These methods are used to manage and update the system tree
  - The root node class is created the following way:

    Root = pyrogue.Root(name='name', description='description')

    - Root classes should be created a sub-class of Root

```
class EvalBoard(pyrogue.Root):
    def __init__(self):
        pyrogue.Root.__init__(self,name='evalBoard',description='Evaluation Board')
```

- Hardware interfaces (pgp, rssi, udp) can be created and linked within the Root class init() method
- Epics and pyro servers can be started within the root class init() method

- Typical top level file utilizing a previously created root class:

```
if __name__ == "__main__":
    evalBoard = EvalBoard()

    appTop = PyQt4.QtGui.QApplication(sys.argv)
    guiTop = pyrogue.gui.GuiTop(group='rogueTest')
    guiTop.addTree(evalBoard)
    appTop.exec_()

    evalboard.stop()
```

PyRogue Root Node

SLAC

- The following attribute is available in the Root class
  - running: A True/False flag indicating if the Root class is running (start() has been called)

- The following methods are provided in the Root class

  - start(pollEn=True, initRead=False, initWrite=False, pyroGroup=None, pyroHost=None, pyroNs=None)
    - Start the Root class and enable external access
    - initRead: Issue a read request to get hardware state
    - initWrite: Issue a write of any default values in the tree
    - pollEn: Set to True to enable system level variable polling
    - The following two args are related to python remote objects (pyro) support which is described later. These can be left at their defaults if pyro is not being used.
      - pyroGroup: enable pyro support with the given group name
      - pyroHost: attached to the passed ip when enabling pyro
      - pyroNs: set the nameserver address

  - stop():
    - Stop the Root class and all of its associated background threads
    - May be overridden in sub-class to stop other support services (epics, mysql) as well

  - addVarListener(func): Add a system level variable listener (described in Variable section)

  - setTimeout(timeout):
    - Set the timeout value for memory access elements in the tree
      - Timeouts are used in memory transactions
      - Timeouts default to 1 second and are usually set higher in hardware simulations

PyRogue Root Node

SLAC

- The following sub-node access method is available
  - getNode(path): Return a node at the full path (evalBoard.AxiVersion.ScratchPad)
- The following variable and command access methods are provided in the Root class
  - get(path): Call the get() method on variable defined by given path
  - getDisp(path): Call the getDisp() method on variable defined by given path
  - value(path): Call the value() method on variable defined by given path
  - valueDisp(path): Call the valueDisp() method on variable defined by given path
  - set(path,value): Call the set(value) method on variable defined by given path
  - setDisp(path,value): Call the setDisp(value) method on variable defined by given path
  - exec(path,arg): Call the call(arg) method on the command defined by given path

Saving And Restoring Configuration

SLAC

- System wide configurations and be saved to or restored from yaml strings using the following methods

- getYaml(readFirst,modes=[‘RW’])
  - Return a yaml string containing values for the entire tree
  - readFirst: True/False flag indication if hardware should be read before generating yaml
  - modes: A list of variable modes which should be included in the yaml dump
    - Configuration saves will use [‘RW’]
    - Status saves will use [‘RO’] or [‘RW’,’RO’]
  - The yaml string contains a dictionary representation of the tree
    - Variable values are saved using getDisp()

- setYaml(yml,writeEach,modes=[‘RW’])
  - Parse the passed yaml string and apply values and execute included commands
  - yml: Yaml string to parse
  - writeEach: True/False flag indicating if each variable should be written as the value is applied
    - If writeEach = False a system level writeAll() operation is executed after the variable values are applied
  - modes: A list of variable modes which should be included in the yaml set operations

- The Root class also supports a stream interface which will generate yaml strings whenever a variable in the system has changed.
  - Allows the root object to be attached to a fileWriter channel
    - Used to embed configuration state in data files

Configuration Array Matching

- In some cases there will be devices or variable that are part of an array:
  - AmcCard[0]
    - DacEnable[0]
    - DacEnable[1]
  - AmcCard[1]
    - DacEnable[0]
    - DacEnable[1]
  - AmcCard[2]
    - DacEnable[0]
    - DacEnable[1]

- In these cases the Yaml configuration file can be setup to apply settings to multiple elements:
  - To Enable Dac channel 0 in both AMCS:

```
AmcCard:
    DacEnable[0]: True
```

  - To Enable both DAC channels in AMC cards 1 and 2

```
AmcCard[1:3]:
    DacEnable: True
```

- Python list slicing indexes are used
  - AmcCard[1:3] accesses AmcCards 1 & 2
  - AmcCard[:2] accesses AmcCards 0 & 1
  - AmcCard[:] accesses all AmcCards
- Additional wildcard features are allowed
  - AmcCard = all cards
  - AmcCard[*] = all cards

Commands & Variables In The Root Node

SLAC

- A number of Command objects are included in the root node:
  - writeAll: Write every variable value to hardware (hidden from the GUI)
  - readAll: read every variable value from hardware
  - writeConfig(path): Store the current configuration to the file passed as an argument
    - Calls getYamlVariables(readFirst=True) and stores the resulting yaml to a file
  - readConfig(path): Apply the configuration from the passed file
    - Reads a yaml string from a file and calls: setOrExecYaml(yml,writeEach=False)
  - hardReset() Issue a system wide hardReset()
  - softReset(): Issue a system wide softReset()
  - countReset(): Issue a system wide countReset()
  - clearLog(): Clear the system log
- A number of Variable objects are included in the rootNode
  - systemLog: A string containing error messages. Allows remote management interfaces to know when an error has occurred. This string contains a summary of the error while the full trace method is dumped to the console.
  - forceWrite: A True/False variable controlling how system level writes are handled. When forceWrite = True all variables are written regardless of their stale state. Otherwise only variables which have seen a set() call since their last write are written.
    - This allows each loaded configuration file to determine how to handle write by include the forceWrite variable at the top of the yaml file.

Special Device Sub-Class: MemoryDevice

SLAC

- A special sub-class of Device, MemoryDevice, provides an alternative to creating a custom Device which uses _rawRead and _rawWrite calls
  - MemoryDevice allows values contained within a yaml configuration file to be written directly to memory, bypassing the normal Variables
    - Usefull for large blocks of data which need to be written
    - Accesses can be non-contiguous
    - A MemoryDevice is write only

- dev = MemoryDevice(name=None, description='', memBase=None, offset=0, size=0, hidden=False, expand=True, enabled=True, base=pyrogue.UInt, wordBitSize=32, stride=4, verify=True)
  - base: base class indicating the type of value contained in the memory
  - wordBitSize: the size of each value in bits
  - stride: the spacing between the value locations in bytes
  - verify: True/False flag enabling verify after write

- The configuration yaml file is expected to contain a set of offsets and values:

```
MemoryDeviceName:
    256: 0x12,0x13,0x14,0x15
    512: 0x22,0x23,0x24,0x25
```

Results in the following address:values pairs being written to memory (wordBitSize=32, stride=4):

256 = 0x12, 260 = 0x13, 264 = 0x14, 268 = 0x15
512 = 0x22, 516 = 0x23, 520 = 0x24, 524 = 0x25

Special Device Sub-Class: RunControl

SLAC

- pyrogue.RunControl is a sub-class of pyrogue.Device which serves as the base class for application specific run control
  - Can be used as is for most software driven data acquisition
  - Can be sub-classed for more complex hardware and external run control

  Go to : https://github.com/slaclab/rogue/blob/master/python/pyrogue/_Device.py around line 253

- The RunControl object is created the following way:

  Rc = pyrogue.RunControl(name, description, hidden=True, rates, states, cmd)

  - Hidden is normally True since RunControl does not appear in the normal GUI tree, instead it is represented in a special window.
  - cmd: Is the command to execute at each iteration when using the default software driven run() method. This can either be a pointer to a python function or pyrogue Command
  - rates: A dictionary of rates and associated keys. The keys are application specific and represent the run rate in hz in the default class. The defaults rates are:

    rates={1:'1 Hz', 10:'10 Hz'}

  - States: A dictionary of states and associated keys. The keys are application specific. The default rates are:

    states={0:'Stopped', 1:'Running'}

Special Device Sub-Class: RunControl

SLAC

- Three methods are used in the RunControl class to control a run

- _setRunState(self,value,changed):
  - This method is called when the runState variable is set
  - The default method checks if the run state has changed
    - A run thread is created and started when going from stopped -> running
    - The run thread is killed and deleted when going from running -> stopped
  - A custom run control class may issue writes to hardware to set the run state

- _setRunRate(self,value):
  - This method is called when the runRate variable is set
  - The default method does nothing
  - A custom run control class may issue a write to hardware to adjust the run rate

- _run():
  - Background method used by the run thread when using the default runState method
  - A custom run control class may modify this method or not use it if setRunState does not start a thread

Special Device Sub-Class: DataWriter

SLAC

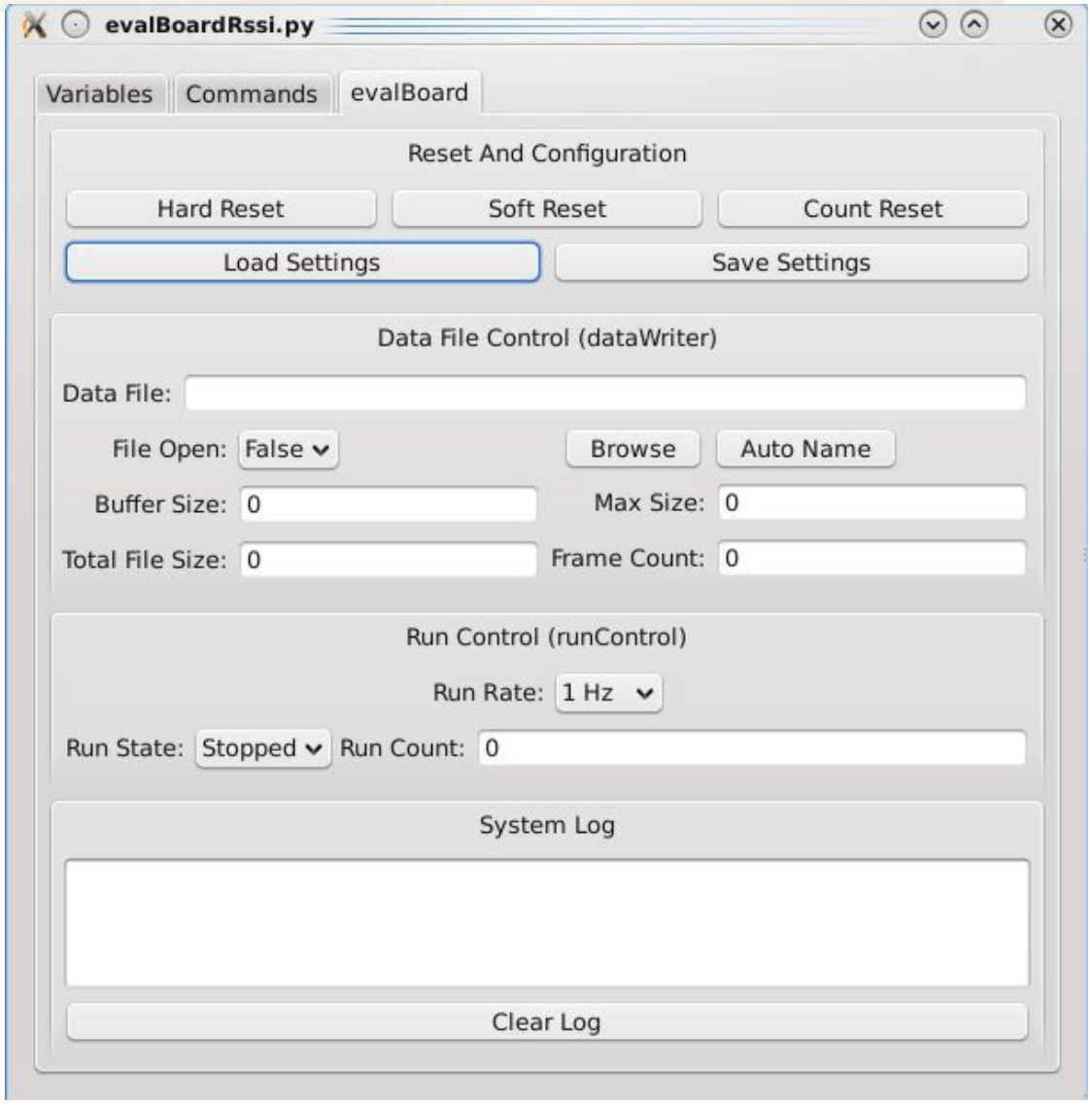
- Similiar to RunControl, DataWriter is a special class which serves at the interface for a file writer in the system
  - pyrogue.utilities.fileio.StreamWriter is the default generic write contained in YAML
  - Custom data writers can be created
- The following methods control the operation of the file writer
  - Can be overridden in a custom writer

    - _setOpen(self,value,changed):
      - Called when the file open/close state is changed
    - _setBufferSize(self,value,changed):
      - Sets the buffer size which sets the amount of data which should be cached before a file system write() is called. Used to avoid small writers. 0 = disable buffering.
    - _setMaxFileSize(self,value,changed):
      - Sets the max file size. A non zero value should force file splitting to avoid large files.
    - _getFileSize(self):
      - Method which should return the current file size
    - _getFrameCount(self):
      - Method which should return the number of frames written

- The following local variables are available:
  - dataFile: Name of data file

- See the StreamWriter Device class as an example:
  - https://github.com/slaclab/rogue/blob/master/python/pyrogue/utilities/fileio.py

RunControl & DataWriter In The GUI



PyRogue Tree Example

SLAC

```
class EvalBoard(pyrogue.Root):
    def __init__(self):
        pyrogue.Root.__init__(self,'evalBoard','Evaluation Board')

        # File writer
        dataWriter = pyrogue.utilities.fileio.StreamWriter('dataWriter')
        self.add(dataWriter)

        # Create the PGP interfaces
        pgpVc0 = rogue.hardware.pgp.PgpCard('/dev/pgpcard_0',0,0) # Registers
        pgpVc1 = rogue.hardware.pgp.PgpCard('/dev/pgpcard_0',0,1) # Data

        # Create and Connect SRP to VC0
        srp = rogue.protocols.srp.SrpV0()
        pyrogue.streamConnectBiDir(pgpVc0,srp)

        # Add configuration stream to file as channel 0
        pyrogue.streamConnect(self,dataWriter.getChannel(0x0))

        # Add data stream to file as channel 1
        pyrogue.streamConnect(pgpVc1,dataWriter.getChannel(0x1))

        # Add Devices
        self.add(surf.axi.AxiVersion(memBase=srp,offset=0x0))
        self.add(surf.protocols.ssi.SsiPrbsTx(memBase=srp,offset=0x30000))

        # Shared memory control interface
        self.smem = pyrogue.smem.SMemControl(group='rogueTest',root=self)
```

PyRogue Tree Example (cont.)

```
        # Run control
        self.add(pyrogue.RunControl(name='runControl' ,
                                    rates={1:'1 Hz', 10:'10 Hz',30:'30 Hz'}))
                                    #cmd=self.SsiPrbsTx.oneShot()))

        # Export remote objects
        self.start(pyroGroup='rogueTest')

        # Create epics node
        self.epics = pyrogue.epics.EpicsCaServer(base='rogueTest',root=self)
        self.epics.start()

    # Override stop method to also stop epics server
    def stop(self):
        self.epics.stop()
        super().stop()
```

Full example code at:

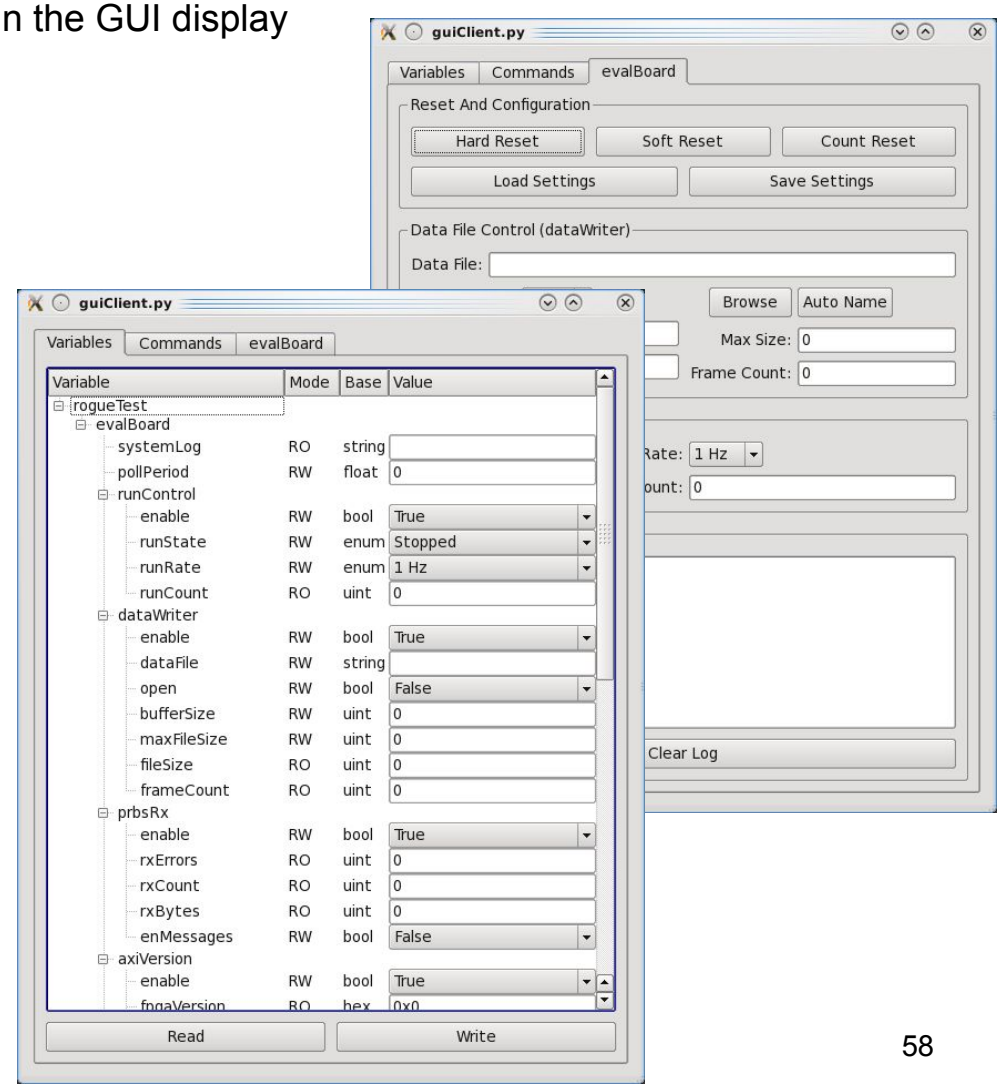
https://github.com/slaclab/rogue-example/blob/master/scripts/evalBoard.py

Adding a GUI

- Pyrogue contains a default GUI application which self populates using the pyrogue tree
- The GUI takes a group name which is used in the GUI display
- Example usage:

```
Import pyrogue.gui
Import PyQt4.QtGui

# Create QT4 application
appTop = PyQt4.QtGui.QApplication(sys.argv)

# Create pyrogue gui and add our tree
guiTop = pyrogue.gui.GuiTop(group='rogueTest')
guiTop.addTree(root=evalBoard)

# Run the qt4 background process
appTop.exec_()

# Stop the eval board after gui exists
evalBoard.stop()
```

The gui may also be attached to a pyro client
Root object to enable remote gui connections



Python Remote Objects (Pyro4)

SLAC

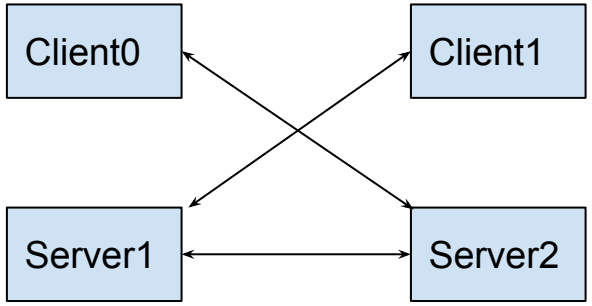
- Most system require a way to access the local server from one or more remote notes
  - One or more remote GUIs
  - Maybe we want to drive the system with python scripts
  - Other control interfaces or a command line

- Pyrogue uses the remote python objects (Pyro4) library to export the management tree for external access.
- Members of the same system identify themselves within a group
  - The mesh software only needs to know the interface and group name
- EPyro export is enabled in the root's start() command

```
evalBoard.start(pyroGroup='rogueTest',pyroHost='127.0.0.1', pyroNs=None)
```

- The pyroHost arg indicates which interface the pyro system should attach to.
- The pyroNs arg indicates the hostname of the nameserver, or None to start one locally
- Rogue makes use of the nameserver feature of Pyro4.
  - Pass pyroNs=None to create one within the server
  - Or start one externally:

```
python3 -m Pyro4.naming --host 127.0.0.1
```



PyRogue Pyro Client: GUI

SLAC

- A client may connect to a pyro4 server using the group and root name:

```
import pyrogue
import pyrogue.gui
import PyQt4.QtGui

client = pyrogue.PyroClient(group='rogueTest',host='127.0.0.1',ns='127.0.0.1')
evalBoard = client.getRoot('evalBoard')

# Create GUI
appTop = PyQt4.QtGui.QApplication(sys.argv)
guiTop = pyrogue.gui.GuiTop(group='rogueTest')
guiTop.addTree(root=evalBoard)

# Run gui
appTop.exec_()
client.stop()
```

PyRogue Pyro Client: Script

- Once a pointer to the root object is obtained using pyro most methods and attributes are accessible just as if you were working on the main root device
- This allows scripts or command line tools (ipython) to interact with the tree remotely

```
import pyrogue
client = pyrogue.PyroClient(group='rogueTest',host='127.0.0.1',ns='127.0.0.1')
evalBoard = client.getRoot('evalBoard')
```

- A user can interact directly with the remote server by executing ipython3 and entering the above text. At this point the user can issue operations:

```
> evalBoard.AxiVersion.FpgaVersion.get()
```

PyRogue Pyro Client: Matlab

- Pyro4 works within matlab with some limitations
  - Some attributes, sub nodes and methods are hidden in matlab and can not be accessed directly. Instead accessor methods must be used:

evalBoard.AxiVersion.ScratchPad.set(10)

must be written as

evalBoard.node(‘AxiVersion’).node(‘ScratchPad’).attr(‘set’,pyargs(‘value’,int32(10)))

- Usage:

Connect to the server:
```
>> client = py.pyrogue.PyroClient(pyargs('group','rogueTest','ns','134.79.229.11'))
>> evalBoard = client.getRoot('evalBoard')
```

Read a value:
```
>> evalBoard.node('AxiVersion').node('ScratchPad').attr('get')

ans = 10
```

Set a value:
```
>> evalBoard.node('AxiVersion').node('ScratchPad').attr('set',pyargs('value',int32(11)))
```

EPICS Channel Access

- Most applications will also need channel access support
- Pyrogue includes a EPICS module for version 3 channel access
- This module generates channel access variables auto learned from the pyrogue tree
- Channel access server can coexist with other management interfaces (pyro, mysql, etc)
- Variables and commands are exported using the pyrogue path value with the '.' replaced with a ':'
  - rogueTest:evalBoard:AxiVersion:FpgaVersion

```
Import pyrogue.epics

# Create epics node
epics = pyrogue.epics.EpicsCaServer(base='rogueTest',root=myRoot)
epics.start()
```

- By default all elements are added to the epics database
- A pvMap can be created to both filter the exported elements and rename them before exporting to epics:

```
# Create epics node
pvMap = {'evalBoard.AxiVersion.UpTimeCnt':'testCnt',
         'evalBoard.AxiVersion.ScratchPad':'testPad'}

self.epics = pyrogue.epics.EpicsCaServer(base='rogueTest',root=self,pvMap=pvMap)
self.epics.start()
```

- Rogue uses the pcaspy package
  - Epics server runs in c++ thread

Shared Memory Access

- A PyRogue tree can be accessed through a shared memory interface
- This is enabled with the following code:

```
self.smem = pyrogue.smem.SMemControl(group='rogueTest',root=self)
```

- A C header is included in the rogue source tree which enables a C or C++ client to interface with the shared memory
- This is required to take state transition commands from a CODA ROC (JLAB DAQ)

Example client code: https://github.com/slaclab/rogue-example/blob/master/src/smem_test.cpp

```
#include <rogue/RogueSMemFunctions.h>
int main (int argc, char **argv) {

   RogueControlMem *smem;
   char res [100];

   if ( rogueSMemControlOpenAndMap(&smem,"rogueTest") < 0 ) {
      printf("Failed to open shared memory\n");
      return(0);
   }

   rogueSMemControlReq(smem,ROGUE_CMD_GET,"evalBoard.AxiVersion.UpTimeCnt","");

   while ( rogueSMemControlAckCheck(smem,res) == 0 ) {
      usleep(100);
   }
   printf("Got %s\n",res);
}
```

Other PyRogue Utilities

- pyrogue.utilities.prbs.PrbsRx
  - Pyrogue Device wrapper around rogue::utilities::Prbs for RX path
  - Exposes Prbs receiver methods as Variables & Commands
  - Allows PrbsRx engine to be accessed from the GUI, mesh nodes and EPICs

- pyrogue.utilities.prbs.PrbsTx
  - Pyrogue Device wrapper around rogue::utilities::Prbs for TX path
  - Exposes Prbs transmitter methods as Variables & Commands

- pyrogue.utilities.fileio.StreamWriter
  - Pyrogue Device wrapper around rogue::utilities::fileio::StreamWriter
  - Exposes StreamWriter methods as Variables and Commands
  - Streamwriter allows multiple streams to be written to a single data file
    - A header identifies each file record, its channel # and Frame flags
  - getChannel(id)
    - Function to expose a slave interface which can be connected to a stream Master
    - Id value is placed in file record header to identify stream

- pyrogue.utilities.fileio.StreamReader
  - Pyrogue Device wrapper around rogue::utilities::fileio::StreamReader
  - Exposes StreamReader methods as Variables and Commands
  - StreamRead acts as a stream source pushing frames as they are read from a data file
    - Channel and flags inserted into Frame flags field

C++ API Wrapper

- The previous portion of this presentation described the management tree which is a pure python construct
  - Unlike the Stream and Memory interfaces which are available in both python and c++
- Rogue contains a set of libraries which create a c++ container around a sub class of Root, making its top level access methods available to a c++ container
- Two interfaces to the wrapper are available
  - Local: Creates an instance of a pyrogue Root class locally and provides access methods

    rogue::ApiWrapperPtr rogue::ApiWrapper::local(std::string module, std::string rootClass)

    - Module is the python module containing the root class
    - rootClass is the root class to access
  - Remote: Attaches to a remote pyrogue instances using the Pyro interface

    rogue::ApiWrapperPtr rogue::ApiWrapper::remote(std::string group, std::string root)

    - group is the pyro group name
    - root is the root instance name
- Data path can be accessed directly by interfacing to driver or by using rogue c++ stream classes

C++ API Wrapper

- The following access methods of the Root class instances are available to the c++ wrapper:
  - get(path): Call the get() method on variable defined by given path
    - A 64-bit integer is returned
  - getDisp(path): Call the getDisp() method on variable defined by given path
    - A string value is returned
  - value(path): Call the value() method on variable defined by given path
    - A 64-bit integer is returned
  - valueDisp(path): Call the valueDisp() method on variable defined by given path
    - A string value is returned
  - set(path,value): Call the set(value) method on variable defined by given path
    - Value is a 64-bit integer
  - setDisp(path,value): Call the setDisp(value) method on variable defined by given path
    - Value is a string
  - exec(path,arg): Call the call(arg) method on the command defined by given path
    - Arg can either be a string or a 64-bit integer
- Additional methods for dealing with additional data types can be added as needed

C++ API Wrapper: Local Example

SLAC

```
#include <rogue/ApiWrapper.h>

int main (int argc, char **argv) {

   try {
      rogue::ApiWrapperPtr wrap = rogue::ApiWrapper::local("evalBoard","EvalBoard");

      printf("Return value int = %li\n",wrap->get("evalBoard.AxiVersion.UpTimeCnt"));
      printf("Return value str = %s\n",wrap->getDisp("evalBoard.AxiVersion.UpTimeCnt").c_str());

      Py_BEGIN_ALLOW_THREADS;
      while(1) {
         usleep(10);
      }
      Py_END_ALLOW_THREADS;

   } catch (boost::python::error_already_set) {
      PyErr_Print();
   }
}
```

- Source code and makefile can be found in the rogue-example project:
  - https://github.com/slaclab/rogue-example/blob/master/src/api_local.cpp

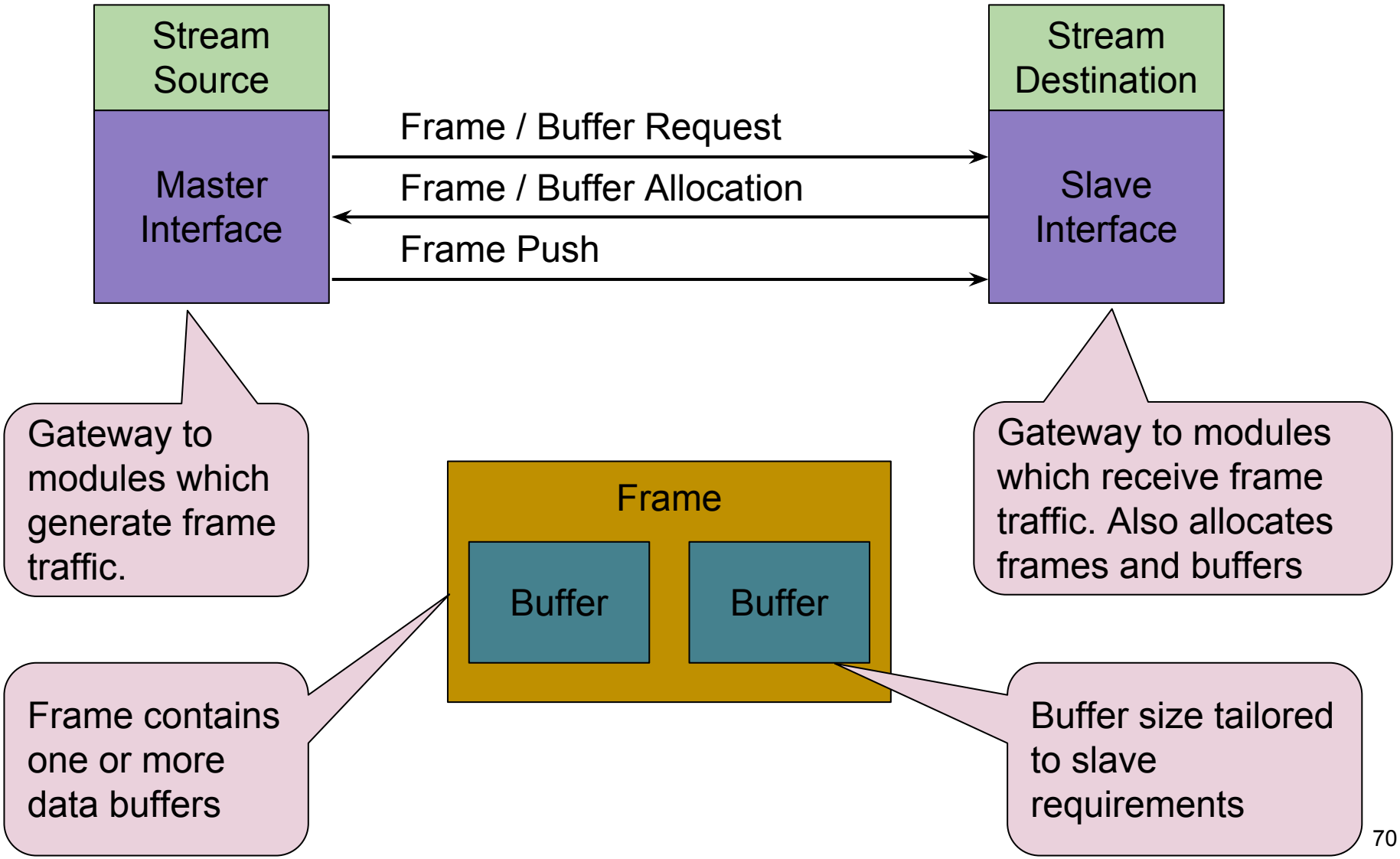
C++ API Wrapper: Remote Example

SLAC

```
#include <rogue/ApiWrapper.h>

int main (int argc, char **argv) {

   try {
      rogue::ApiWrapperPtr wrap = rogue::ApiWrapper::remote("rogueTest","evalBoard");

      printf("Return value int = %li\n",wrap->get("evalBoard.AxiVersion.UpTimeCnt"));
      printf("Return value str = %s\n",wrap->getDisp("evalBoard.AxiVersion.UpTimeCnt").c_str());

   } catch (boost::python::error_already_set) {
      PyErr_Print();
   }
}
```

- Source code and makefile can be found in the rogue-example project:
  - https://github.com/slaclab/rogue-example/blob/master/src/api_remote.cpp

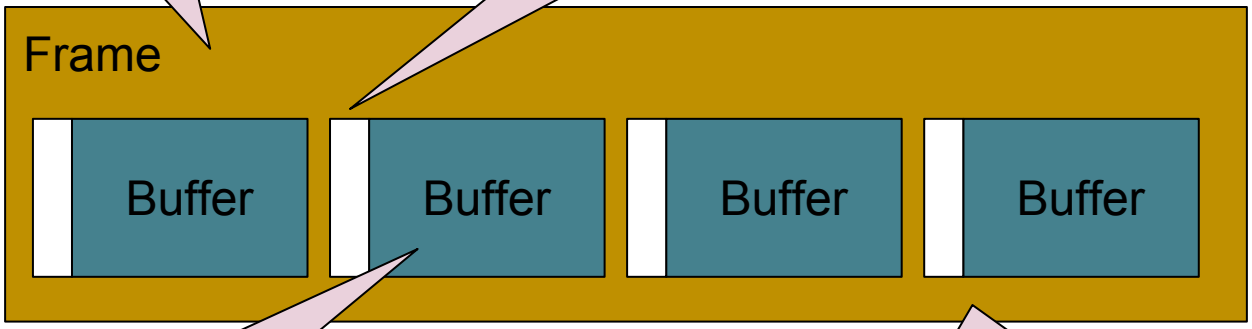
Stream Interface



Frames And Buffers

Frame container holds a set of buffers which contain payload data

Each buffer is split between ‘header’ and ‘payload’ data. Only payload data is available using the frame API



Size of buffers is determined by the interface transaction size limit and by the included protocols

Some modules will access raw buffer data (UDP, PGP, AXI-Stream) while higher level clients (file IO & data processing) will use Frame API

Frame API

- A frame is a container for a string of buffers which contain the actual data.
- Frame API is available to both C++ and python
  - getAvailable()
    - Get available capacity of the frame in bytes (size - data already written)
  - getPayload()
    - The the payload size of the frame in bytes (not including hidden headers)
- Read and Write frame data in C++:
  - uint32_t ris::Frame::write (void *p, uint32_t offset, uint32_t count)
    - Write count bytes from buffer p into the frame.
    - Offset sets the location within the frame
    - Returns the number of byte written, an exception is thrown if write exceeds available size
  - uint32_t ris::Frame::read (void *p, uint32_t offset, uint32_t count)
    - Read count bytes from the frame into buffer p.
    - Offset sets the location within the frame
    - Returns the number of bytes read, an exception is thrown if read exceeds available data

Frame API

- Read and Write frame data in Python:
  - write (ba, uint32_t offset)
    - Write data from passed byte array into frame.
    - Offset sets the location within the frame.
    - Size is determined by the byte array size
    - An exception is thrown if write exceeds available size
  - read (ba, uint32_t offset)
    - Read from the frame and insert it into the passed byte array into frame.
    - Offset sets the location within the frame
    - Size is determined by the byte array size
    - An exception is thrown if read exceeds available data

Stream Master API

SLAC

- The stream Master API is used to send a frame from one module to another.
  - Frame master and slaves written both in c++ and Python can be intermixed.

- A master but be connected to at least one slave before a frame can be send
  - void setSlave(rogue::interfaces::stream::SlavePtr)
    - Exposed as _setSlave in python
    - Connects a slave interface instance to this master.
    - Used to set the ‘primary’ slave interface which will accept frame / buffer requests.
    - The primary slave will always receive the frame last.

- A secondary slave can also be attached to a master.
  - addSlave(rogue::interfaces::stream::SlavePtr)
    - Exposed as _addSlave in python
    - Adds a ‘secondary’ slave interface instance to this master.
    - The secondary interfaces will receive the frame before the primary.

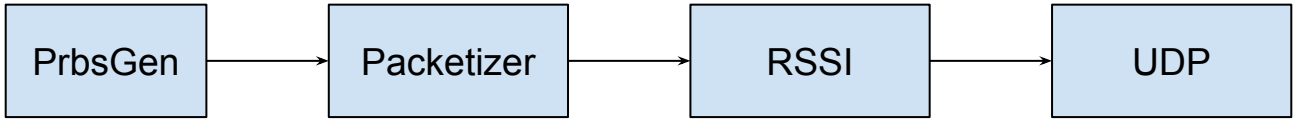
- The following python helpers are available:
  - streamConnect(source,dest):
    - Connect a primary slave to a master: source.setSlave(dest)
  - streamTap(source,dest):
    - Connect a secondary slave to a master: source.addSlave(dest)
  - streamConnectBitDir(A,B):
    - Connect A and B in a bi-directional way
      - streamConnect(A,B)
      - streamConnect(B,A)

Stream Master API

- Before a frame can be send from a master to a slave, a new frame must be created
  - The master will issue a frame request from to the slave.
  - The slave will allocate a new frame (possibly by requesting buffers from a kernel driver)

  - frame = master.reqFrame(size, zeroCopyEn, maxBufferSize)
    - Request a frame with total payload size of ‘size’
    - zeroCopyEn: Determined if the requester can accept a zero copy buffer. Usually = True.
    - maxBufferSize: sets the max size of each buffer within the frame. Pass zero for unlimited.

  - Intermediate stream interface nodes will forward the request to the last slave in the chain.
    - Size will be adjusted along the way if additional header space is required
    - zeroCopyEn and maxBufferSize may also be adjusted



- Sending a frame:
  - sendFrame(frame)
    - Exposed as _sendFrame in python
    - Push a frame to the attached slaves.

- Frames are automatically de-allocated when they are not longer referenced

Example Stream Master In Python

SLAC

```
import rogue.interfaces.stream

class frameTransmit(rogue.interfaces.stream.Master):

        def __init__(self):
          rogue.interfaces.stream.Master.__init__(self)

        def generateFrame(self):
          # Request 1024 byte frame, zeroCopyEn, no limit on buffer size
          frame = self._reqFrame(1024,True,0)

          # Create a byte array for frame data
          ba = bytearray(frame.getAvailable())

          # Set byte array contents
          for x in range(frame.getAvailable()):
              ba[x] = x

          # Copy byte array to frame at offset 0
          frame.write(ba,0)

          # Send frame
          self._sendFrame(frame)
```

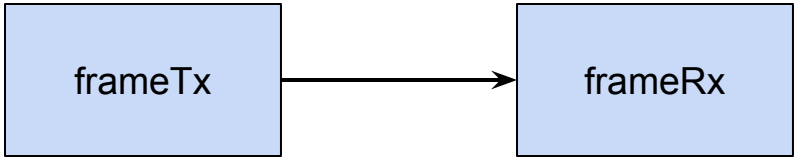
Stream Slave API

- Frames received by a slave module generate a callback method ‘acceptFrame’
- acceptFrame(frame)
  - Exposed as _acceptFrame in python
  - Accept a frame push from the master class
  - Override in sub-class to receive frame and work with frame data
- Example python Slave:

```
Import rogue.interfaces.stream

class frameReceive(rogue.interfaces.stream.Slave):

   def __init__(self):
      rogue.interfaces.stream.Slave.__init__(self)

   def _acceptFrame(self,frame):
      print("Got Frame size=%i" % (frame.getPayload()))
```

Python Streaming Example

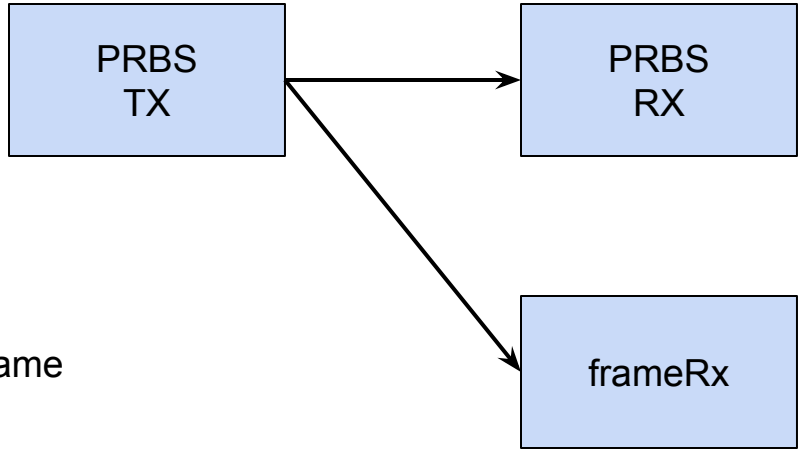
- Using classes from previous slides:




```
rx = frameReceive()
tx = frameTransmit()

pyrogue.streamConnect(tx,rx))

while (True):
    tx.generateFrame()
```

- Using PRBS module and receive class from previous slide:

```
Include rogue.utilities

prbsTx = rogue.utilities.Prbs()
prbsRx = rogue.utilities.Prbs()
rx = frameReceive()

pyrogue.streamConnect(prbsTx,prbsRx)
pyrogue.streamTap(prbsTx,rx)

while(True):
   prbsTx.generate(100)   # Generate a 100 byte frame
```



Common Python Streaming EndPoints

SLAC

- rogue.hardware.pgp.pgpCard(path, lane, vc)
  - Create a pgp stream master/slave for given lane & vc
    pgpVc0 = pgpCard('/dev/pgpcard_0', 0, 0)

- rogue.hardware.rce.axiStream(path, dest)
  - Create an axiStream (RCE) stream master/slave for given destination
    rce2 = axiStream('/dev/axistream_0', 2)

- rogue.hardware.data.dataCard(path, dest)
  - Create a data development card stream master/slave for given destination
    data3 = dataCard('/dev/datadev_0', 2)

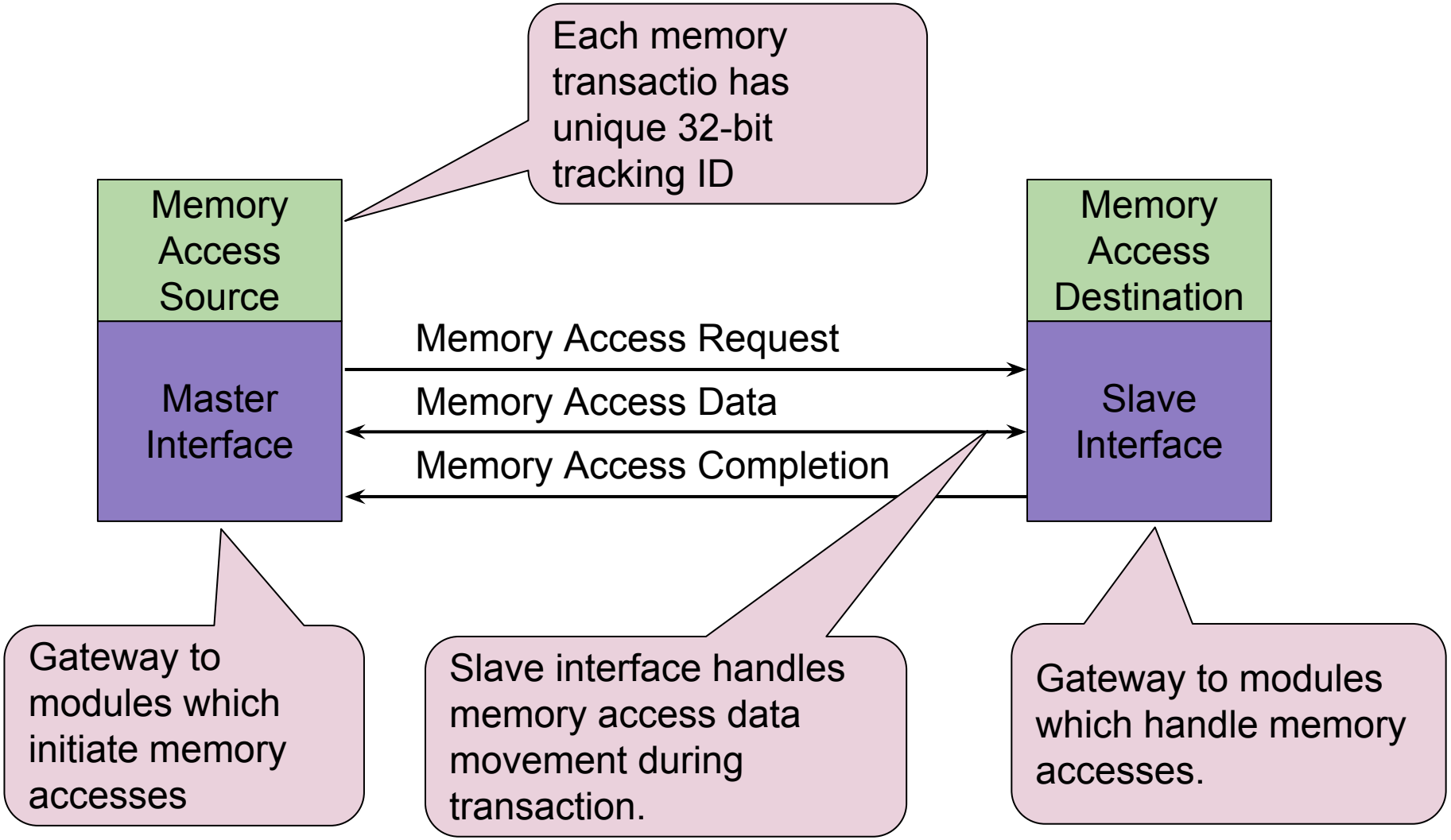
- pyrogue.protocols.UdpRssiPack(host,port,size,wait=True)
  - Create a combined UDP / RSSI / Packetizer master/slave for given ip, port & MTU
    udp = pyrogue.protocols.UdpRssiPack(host='192.168.2.194',port=8192,size=1400)

  - The wait flag indicates if the interface creation should block until the connection is made
  - The UdpRssiPack class is a Device wrapper around the three C++ classes
    - Exposes debug counters in LocalVariables
      https://github.com/slaclab/rogue/blob/master/python/pyrogue/protocols.py

  - Use the applications(channel) method to access a specific packetizer channel:
    srp = rogue.protocols.srp.SrpV3()
    pyrogue.streamConnectBiDir(srp,udp.application(0))

  - UDP without RSSI for given ip, port & MTU
    udp = rogue.protocols.udp.Client(host='192.168.2.194',port=8192,size=1400)

Memory Interface


Each memory transactio has unique 32-bit tracking ID
Memory Access Source
Master Interface
Memory Access Request
Memory Access Data
Memory Access Completion
Memory Access Destination
Slave Interface
Gateway to modules which initiate memory accesses
Slave interface handles memory access data movement during transaction.
Gateway to modules which handle memory accesses.

Common Memory Slaves

- The most common memory slaves used are the SRPV0 and SRPV3 gateways
  - rogue.protocols.srp.SrpV0()

```
srp = rogue.protocols.srp.SrpV0()
pyrogue.streamConnectBiDir(srp,udp.application(0))
```

  - rogue.protocols.srp.SrpV3()

```
srp = rogue.protocols.srp.SrpV3()
pyrogue.streamConnectBiDir(srp,udp.application(0))
```

- The command protocol used in some camera electronics is also available:
  - rogue.protocols.srp.Cmd()

```
Cmd = rogue.protocols.srp.Cmd()
cmd.sendCmd(opCode=0x5, context=0xAA)
```

Dummy Memory Slave

SLAC

- Sometimes software testing has to be done without real hardware
  - A good chunk or rogue core development
  - Testing new hardware interface classes

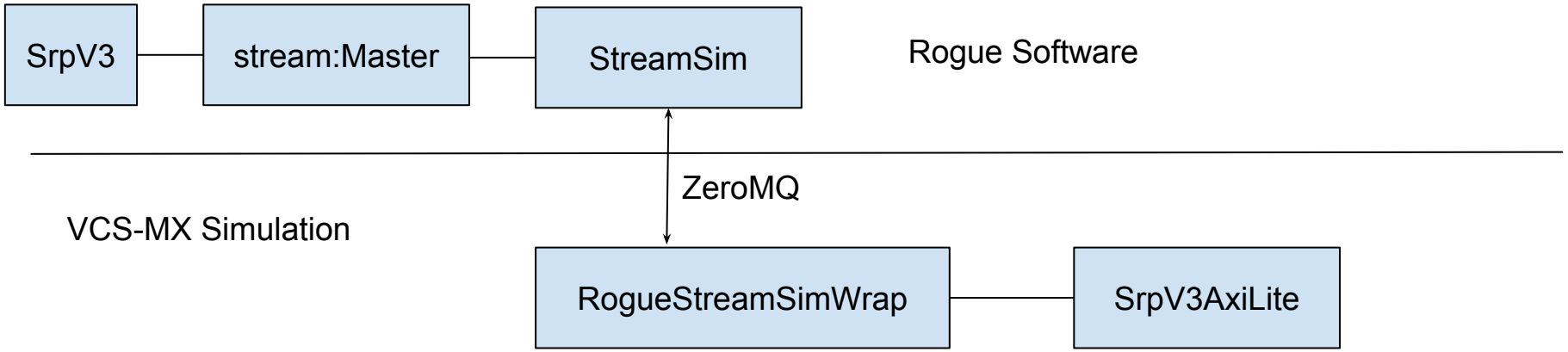
- The pyrogue.simulation.MemEmulate() class is a sub-class of memory slave written in c++ which emulates hardware
  - Memory space is allocated as it is accessed, and initialized to 0
  - Reads from an address will return the last value written
  - minTransaction and maxTransaction are accepted as arguments
    - Usually set to match the SRP version being used

Example:

```
srp = pyrogue.simulation.MemEmulate(minWidth=4,maxSize=4096)

# Add Devices
self.add(surf.axi.AxiVersion(memBase=srp,offset=0x0))
self.add(surf.protocols.ssi.SsiPrbsTx(memBase=srp,offset=0x30000))

self.start(pyroGroup='rogueTest')
```

Firmware Simulation Interface

- Rogue provides the ability to interface to a firmware simulation as if it was a talking to real hardware
  - pyrogue.simulation.StreamSim provides a streaming interface between a rogue stream master/slave and an AxiStream simulation entity in firmware
  - Allows production rogue software to run against production firmware before compile
    - Network interface or PGP is replaced with stream simulation link



Firmware Simulation Interface

SLAC

- The StreamSim client interface in Rogue is created in the following way
  - Rogue is started after the hardware simulation is started

    streamSim = pyrogue.simulation.StreamSim(host='localHost',dest=0,uid=1,ssi=True)

    - host = the host to connect to. This is the host running the vcs simulation
    - dest = the destination channel in the simulation to attach to
    - uid = A unique session id, arbitrarily defined by the user
    - Ssi = emulate an SSI link (Use SOF and EOFE, usually True)

Example Code:

```
srp = pyrogue.simulation.StreamSim(host='localHost',dest=0,uid=1,ssi=True)

# Add Devices
self.add(surf.axi.AxiVersion(memBase=srp,offset=0x0))
self.add(surf.protocols.ssi.SsiPrbsTx(memBase=srp,offset=0x30000))

self.start(pyroGroup='rogueTest')
```

Firmware Simulation Interface

- The VHDL module used in the simulator has the following template:

```
entity RogueStreamSimWrap is
   generic (
      TPD_G                        : time                         := 1 ns;
      DEST_ID_G                    : integer range 0 to 255   := 1;
      USER_ID_G                    : integer range 0 to 100   := 1;
      COMMON_MASTER_CLK_G  : boolean                    := false;
      COMMON_SLAVE_CLK_G     : boolean                    := false;
      AXIS_CONFIG_G                : AxiStreamConfigType    := AXI_STREAM_CONFIG_INIT_C
      );
   port (

      -- Main Clock and reset used internally
      clk : in sl;
      rst : in sl;

      -- Slave
      sAxisClk    : in  sl;            -- Set COMMON_SLAVE_CLK_G if same as clk input
      sAxisRst    : in  sl;
      sAxisMaster : in  AxiStreamMasterType;
      sAxisSlave  : out AxiStreamSlaveType;

      -- Master
      mAxisClk    : in  sl;            -- Set COMMON_MASTER_CLK_G if same as clk input
      mAxisRst    : in  sl;
      mAxisMaster : out AxiStreamMasterType;
      mAxisSlave  : in  AxiStreamSlaveType;

      -- Sideband Clock and reset, clocked with clk and rst
      opCode      : out slv(7 downto 0);
      opCodeEn    : out sl;
      remData     : out slv(7 downto 0)
      );
end RogueStreamSimWrap;
```

A Few Closing Notes

SLAC

- rogue/Pyrogue is meant to be extremely extensible both in the python and c++ layers
  - If a common Class does not fit your needs, please sub class it!
  - If you feel your sub-class, feature or helper function would be useful to other users, please consider adding it to the core
    - We will work with you to find the proper namespace for this feature
      - Jesus's streaming to LocalVariable gateway class is an example of this

- Don't be afraid of the C++ layers and libraries
  - Rogue is flexible enough to allow porting of modules written first in python to the c++ layer to enhance performance
  - There are still a few modules which would be useful
    - Data processing template class in c++
      - To accelerate data processing libraries in the c++ layer
    - Streaming buffer gateway to queue stream data (with optional copy) to allow slow processes to exist without slowing down the data processing threads

- Ask questions!
  - There are some features of rogue that exist purely because someone asked:

"How can I do this?" or "It would be really cool if I could do this"

Helpful Links

SLAC

- Rogue is on github:
  - https://github.com/slaclab/rogue
- SLAC email list for rogue announcements:
  - https://listserv.slac.stanford.edu/cgi-bin/wa?A0=AIR-ROGUE-USERS
  - Please subscribe if you are a user of rogue
- Rogue-daq channel on SLACK
  - https://slac.slack.com/#rogue-daq
  - Developers are usually available
- SLAC JIRA for the rogue project
  - https://jira.slac.stanford.edu/projects/ESROGUE/summary
  - Please submit tickets if you are seeing any bugs
- Questions:
  - Either find us on slack or email:
    - rherbst@slac.stanford.edu
    - bareese@slac.stanford.edu